Australian Government
Department of Defence
Defence Science and
Technology Organisation

Mesh Generation via Local Bisection Refinement of Triangulated Grids

Jason R. Looker

Joint and Operations Analysis Division

Defence Science and Technology Organisation

DSTO–TR–3095

ABSTRACT

This report provides a comprehensive implementation of an unstructured mesh generation method that refines a triangulated grid by locally bisecting triangles on their longest edge, until they satisfy a given local condition. The method is relatively simple to implement, has the capacity to quickly generate a refined mesh with triangles that rapidly change size over a short distance, and does not create triangles with small or large angles.

APPROVED FOR PUBLIC RELEASE

Published by

DSTO Defence Science and Technology Organisation
506 Lorimer St,
Fishermans Bend, Victoria 3207, Australia

Telephone: 1300 333 362
Facsimile: (03) 9626 7999

© *Commonwealth of Australia 2015*
AR No. AR 016–273
June 2015

APPROVED FOR PUBLIC RELEASE

Mesh Generation via Local Bisection Refinement of Triangulated Grids

Executive Summary

A mesh is a collection of polygonal (or polyhedral) elements that are designed to approximate a geometric domain. Meshes have numerous and varied applications in engineering and scientific computing, such as the numerical solution of partial differential equations using the finite element method, computer aided design, rendering computer graphics, and for representing terrain surfaces in geographical information systems.

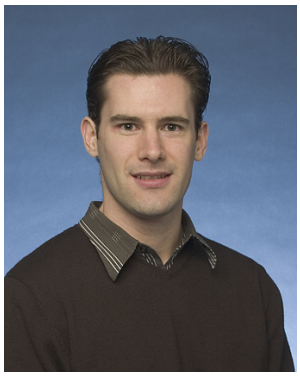
This report emerged from a study of path planning for military aircraft conducting missions in hostile environments, where the objective was to control the aircraft such that the risk posed by the environment was minimised. In that study, mesh elements were smallest in regions where the risk to the aircraft was highest. This enabled minimum risk paths to be accurately captured while restricting the number of elements in the mesh, thus improving the efficiency of the numerical method used to calculate optimal paths.

This report documents a comprehensive implementation of an unstructured mesh generation method that refines a triangulated grid by locally bisecting triangles on their longest edge until they satisfy a given local condition. The refined mesh consists of regular right-angled triangles, which have 45, 45 and 90 degree angles. The vertices of the triangles are ordered such that minimal computations are required to identify their longest edge and to ensure the bisected triangles are compatible, that is, all edges are shared with at most one other triangle. The method is relatively simple to implement, has the capacity to quickly generate a refined mesh with triangles that rapidly change size over a short distance, and does not create triangles with small or large angles.

The implementation provided by this report has the following desirable features. Mesh data structures are chosen to enable local bisection refinement to occur in constant running time and with minimal computations. A local refinement condition is derived that guarantees the local triangulation diameter of the refined mesh will obey a specified bound; hence the numerical error of computations on the mesh can be controlled while restricting the number of triangles. The mesh refinement algorithm is tested and shown to achieve the anticipated linear running time with respect to the number of triangles in the refined mesh. Employing a uniform triangulated grid to initialise the mesh creates virtual buckets. It follows that point location can be accomplished in constant running time without requiring additional memory.

THIS PAGE IS INTENTIONALLY BLANK

Author



Jason Looker
Joint and Operations Analysis Division

Dr Jason Looker joined DSTO in 2006 as an Operations Research Scientist after completing a BSc (Honours) and PhD in Mathematics at The University of Melbourne. He has undertaken operations analysis in support of Air Force Headquarters and Project AIR 9000 Phase 8 (Naval Combat Helicopter), and is leading a research project on the path planning of military aircraft through hostile environments.

THIS PAGE IS INTENTIONALLY BLANK

Contents

Notation **xi**

1 Introduction **1**

1.1 Local Bisection Refinement 2

1.2 Triangulation Diameter 4

2 Mesh Data Structures **5**

2.1 Attributes 5

2.2 Memory Allocation 7

3 Local Refinement Condition **9**

4 Coarse Triangulation **10**

4.1 Construction 10

4.2 Mesh Initialisation 14

5 Mesh Refinement **15**

5.1 Bisecting Triangles 15

5.2 Updating Triangle Neighbours 18

5.3 Refinement Algorithm 20

5.4 Computational Performance 21

6 Adjacency List **22**

7 Point Location **23**

7.1 Initial Triangle Selection 24

7.2 Triangle Containing the Query Point 25

8 Conclusion **28**

References **29**

Appendices

A Domains other than $[0,1]^2$ **30**

B Other Cases for Bisecting Triangles and Updating their Neighbours **33**

Figures

1 A computational mesh generated using Maubach's method. 2

2 A coarse triangulation showing the assignment of indices to nodes and triangles, and the grid spacing δ. 11

3 The configuration of the descendants of t_i for the case $k(t_i) = 1$ and $O(t_i) = 0$. The references to the descendants are i and $\bar{i}$, and a, b, c and d denote references to vertices. Note that $N_{\mathbf{x}}$ and N_t represent their states immediately prior to bisecting t_i. 16

4 The configuration of triangle neighbours before and after the bisection of compatibly divisible triangles t_i and t_{i_b} for the case $k(t_i) = 1$, $O(t_i) = 0$ and $O(t_{i_b}) = 0$. Here a, b, c and d represent triangle neighbours. 18

5 The scaled average running time of the implementation of Maubach's method documented in this report, versus the scaled number of triangles; a line of best fit is shown with the data points. 22

6 The orientation test: does d lie on, to the left of, or to the right of the line defined by the triangle edge (a, b)? The arrows indicate the directions of the vectors used to perform the test. The test is applied to the other edges in the same manner. 26

THIS PAGE IS INTENTIONALLY BLANK

Notation

$\mathbb{N}$	natural numbers
$\mathbb{Z}^n$	n-dimensional space of integers
$\mathbb{R}^n$	n-dimensional space of real numbers
$\mathbf{x}$	a generic n-dimensional vector, $\mathbf{x} = (x_1, x_2, \ldots, x_n)$
$\|\mathbf{x}\|$	magnitude (two-norm) of $\mathbf{x}$, $\|\mathbf{x}\| = \sqrt{\sum_{i=1}^{n} x_i^2}$
$\|\mathbf{x}\|_1$	one-norm of $\mathbf{x}$, $\|\mathbf{x}\|_1 = \sum_{i=1}^{n} \|x_i\|$
$(e_1, e_2, \ldots)$	an ordered list with elements e_i; for example, a vector
$\{e_1, e_2, \ldots\}$	an unordered list with elements e_i; for example, a set
$\|\cdot\|$	number of entries in a data structure
$a = b \mod i$	modular arithmetic
$\lceil \cdot \rceil$	ceiling of a real number
$\lfloor \cdot \rfloor$	floor of a real number
$\mathcal{O}(\cdot)$	"big O" Landau symbol
Ω	the given domain, $\Omega \subset \mathbb{R}^2$
t	triangle in Ω
T	triangulation of Ω
$\mathscr{V}(T)$	set of nodes that comprise T
$N_{\mathbf{x}}$	number of nodes in T
N_t	number of triangles in T
$\overline{\delta}$	the specified maximum grid spacing for T
h	triangulation diameter
$h(\mathbf{x})$	local triangulation diameter at $\mathbf{x} \in \mathscr{V}(T)$
$\overline{h}$	the specified target triangulation diameter
$\overline{h}(\mathbf{x})$	the specified target local triangulation diameter at $\mathbf{x} \in \mathscr{V}(T)$
$\overline{h}(t)$	the specified target local triangulation diameter at $t \in T$
$\overline{h}(i)$	the specified target local triangulation diameter at $t_i \in T$
T_0	coarse triangulation of Ω
δ	grid spacing of T_0
t_b	bisection neighbour of t
i_b	index of the bisection neighbour of t_i
ℓ_b	bisection edge length
ℓ_b^{-1}	inverse bisection edge length

b_t	the specified target bisection edge length for t
$L(t)$	level of refinement of t
$O(t)$	orientation of the arrangement of the vertices of t
$V(t)$	references to the vertices of t, $V(t) = (v_1, v_2, v_3)$
$N(t)$	references to the neighbours of t, $N(t) = (n_1, n_2, n_3)$
$\mathcal{N}$	node data structure (array)
$\mathcal{T}$	triangle data structure (array)
$\mathcal{D}$	generic data structure (array)
$\mathcal{D}[i]$	ith entry in $\mathcal{D}$
$\mathcal{D}[i, j]$	jth entry in $\mathcal{D}[i]$
$\mathcal{D}[i : j]$	ith to jth consecutive entries in $\mathcal{D}$
$\mathcal{D}[i, j : k]$	jth to kth consecutive entries in $\mathcal{D}[i]$
$\mathcal{T}[i, 1 : 3]$	$V(t_i)$
$\mathcal{T}[i, 4 : 6]$	$N(t_i)$
$\mathcal{T}[i, 7]$	$O(t_i)$
$\mathcal{T}[i, 8]$	$L(t_i)$
$\mathcal{P}_\mathcal{N}$	null data array for padding $\mathcal{N}$
$\mathcal{P}_\mathcal{T}$	null data array for padding $\mathcal{T}$
`True`	"true" Boolean data type
`False`	"false" Boolean data type
$\mathscr{N}(\mathbf{x})$	neighbourhood of $\mathbf{x} \in \mathscr{V}(T)$
N_e	number of triangles in T_0 with an edge on any one side of $[0, 1]^2$
$\mathcal{T}^\mathrm{V}$	data structure for references to triangle vertices in T
$\mathcal{T}_0^\mathrm{V}$	data structure for references to triangle vertices in T_0
$\mathcal{T}_0^\mathrm{N}$	data structure for references to triangle neighbours in T_0
β	reference to a null boundary triangle, $\beta < 0$
$k(t_i)$ and $k(i)$	determine which vertices of t_i form the bisection edge
$\mathcal{S}_b$	stack (LIFO queue) of references to triangles to bisect
$\mathcal{A}$	adjacency list data structure
$\mathcal{E}[i]$	edges of t_i
M_L	maximum level of refinement in T

1 Introduction

A mesh is a collection of polygonal (or polyhedral) elements that are designed to approximate a geometric domain. Meshes have numerous and varied applications in engineering and scientific computing, such as the numerical solution of partial differential equations using the finite element method, computer aided design, rendering computer graphics, and for representing terrain surfaces in geographical information systems.

This report emerged from a study of path planning for military aircraft conducting missions in hostile environments, where the objective was to control the aircraft such that the risk posed by the environment was minimised. In that study, mesh elements were smallest in regions where the risk to the aircraft was highest. This enabled minimum risk paths to be accurately captured while restricting the number of elements in the mesh, thus improving the efficiency of the numerical method used to calculate optimal paths [Looker 2013].

Meshes are either structured or unstructured. A node (element vertex) of a structured mesh is referenced by a tuple of indices (one for each spatial dimension), and adjacent nodes are found by translating each index of the tuple by unity. A node of an unstructured mesh is referenced by a single index, and topological attributes of the mesh are stored in separate data structures to enable adjacent nodes and elements to be found [Hjelle & Dæhlen 2006]. Note that the characterisation of a mesh as either structured or unstructured relates to how the mesh is stored in memory; it does not relate to the geometric structure of the mesh. However, structured meshes must have an underlying geometric structure, whereas this may or may not be the case for unstructured meshes [Nelson 2014].

Structured meshes are simpler to implement and faster to generate compared with unstructured meshes. However, unstructured meshes allow greater flexibility in the choice of element, and can attain a rapid change in element size over a shorter distance using fewer elements than a structured mesh, resulting in more efficient computations on the mesh [Shewchuk 1997*b*, Shewchuk 2012, Nelson 2014].

This report documents a comprehensive implementation of the unstructured mesh generation method of Maubach [1995], focussing on the case of a two-dimensional mesh; although the method can be applied in higher dimensions [Maubach 1995, Arnold, Mukherjee & Pouly 2000]. Maubach's method refines a triangulation (a mesh with triangular elements) by locally bisecting triangles on their longest edge until they satisfy a given local condition. The refined mesh consists of regular right-angled triangles, which have 45, 45 and 90 degree angles, as shown in Figure 1. The vertices of the triangles are ordered to allow their longest edge to be identified without any computations[1] and to ensure the bisected triangles are compatible, that is, all edges are shared with at most one other triangle.

Maubach's method is relatively simple to implement, has the capacity to quickly generate a refined mesh with triangles that rapidly change size over a short distance, and does not create triangles with small or large angles (which may cause numerical difficulties). However, all refined elements are regular right-angled triangles and this may be too

[1]The phrase "without any computations" is to be interpreted as "without any searching, sorting or numerical computations" throughout this report.

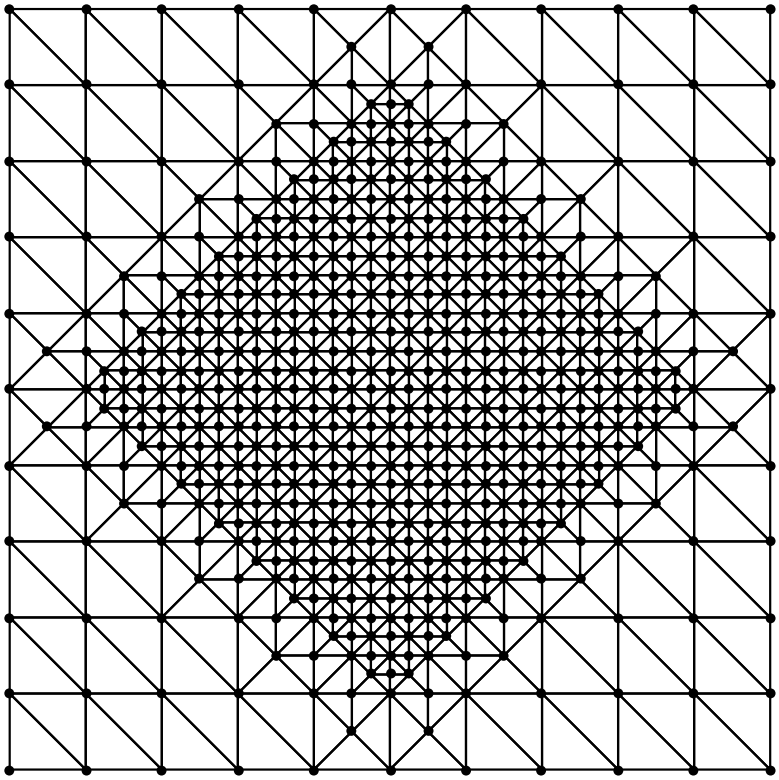


Figure 1: *A computational mesh generated using Maubach's method.*

restrictive for some applications [Shewchuk 2012]. Furthermore, other techniques such as Delaunay mesh generation [Cheng, Dey & Shewchuk 2012] produce size-optimal meshes that have fewer triangles without losing numerical precision, and hence more efficient computations can be performed on the resulting mesh.

The introduction continues with a review of local bisection refinement where the key algorithms of Maubach's method are presented, and concludes with a brief discussion of the triangulation diameter. Section 2 examines the data structures chosen to store the mesh. A condition for local mesh refinement is derived in Section 3 with the aim of controlling the numerical error of computations performed on the mesh. Initial mesh construction for the case of a uniform triangulated grid is presented in Section 4. Section 5 contains the implementation of Maubach's method that is the principal subject of this report. An algorithm for constructing an adjacency list of the nodes from the refined mesh is presented in Section 6. Determining a triangle that contains a given point is a fundamental operation on a mesh, and is known as point location; this is discussed in Section 7.

1.1 Local Bisection Refinement

In this section, some elementary notation and terminology is introduced, followed by a review of local bisection refinement.

Let $\Omega \subset \mathbb{R}^2$ be a given domain. A triangle in Ω is denoted by t where $\mathbf{x}_i \in \Omega$ are the vertices of t. A triangulation T of Ω is defined by[2]

$$T = \{t \mid t \text{ do not overlap, are compatible, and all } \mathbf{x}_i \in \Omega\}.$$

The nodes of T are defined to be the set of vertices that comprise all $t \in T$. A more precise definition of T can be found in Hjelle & Dæhlen [2006].

Maubach's method begins with a coarse triangulation T_0 of Ω, where all $t_0 \in T_0$ are identical. The number of bisections of t_0 required to generate a descendant $t \in T$ is the level of refinement of t, given by $L(t)$; the level of refinement of all t_0 is defined to be zero. Maubach's method generates a refined T by locally bisecting triangles on their longest edge, which shall be referred to as the bisection edge.

Local bisection refinement is accomplished by two procedures: the first bisects a triangle and maintains an ordering of the descendant's vertices to enable their bisection edges to be subsequently identified; and the second procedure guarantees the descendants are compatible.

Initially all $t \in T_0$ have their vertices arranged anticlockwise such that the first and third vertices form the bisection edge. `BisectTriangle` (Algorithm 1) bisects a given triangle t and creates descendants t_1 and t_2. The order of the vertices of t_1 and t_2 is chosen to ensure their bisection edges can be immediately identified, without any computations, if they are subsequently bisected. However, `BisectTriangle` is not sufficient to generate a refined T as mesh incompatibilities will be introduced.

Algorithm 1: `BisectTriangle`

Input: $t \in T$ where t has vertices $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$
Result: descendants t_1 and t_2 are created

```
1  k ← 2 − (L(t) mod 2)
2  z ← ½ (x_1 + x_{k+1})
3  if k = 1 then
4  |   t_1 ← (x_1, z, x_3)
5  |   t_2 ← (x_2, z, x_3)
6  else
7  |   t_1 ← (x_1, x_2, z)
8  |   t_2 ← (x_2, x_3, z)
9  L(t_1) ← L(t) + 1
10 L(t_2) ← L(t) + 1
```

A record of triangle neighbours is maintained to facilitate the avoidance of mesh incompatibilities. Two triangles are neighbours if they share an edge. Compatibly divisible triangles are neighbours that share their bisection edge. The neighbour of t on its bisection edge is called the bisection neighbour and denoted by t_b. Incompatibilities are avoided if two compatibly divisible triangles are bisected simultaneously. If $L(t) = L(t_b)$ then t

[2]The symbol T may denote a generic triangulation or the state of a triangulation being refined using Maubach's method.

and t_b are compatibly divisible, otherwise t will be compatibly divisible with one of the descendants of t_b [Maubach 1995, Theorem 5.1].

This is exploited in the recursive algorithm `RefineTriangle` (Algorithm 2) to compatibly refine a given triangle; the recursion depth of `RefineTriangle` is bounded by the maximum level of refinement in T [Maubach 1995]. `RefineTriangle` calls itself repeatedly on a sequence of triangles until a compatibly divisible triangle is found. This sequence of triangles is then bisected in reverse order to preserve compatibility.

Algorithm 2: `RefineTriangle`

Input: $t \in T$
Result: t is compatibly refined

```
1  t_b ← bisection neighbour of t
2  if L(t) = L(t_b) then
3  |  BisectTriangle(t)
4  |  BisectTriangle(t_b)
5  |  foreach compatibly divisible triangle t' visited by RefineTriangle do
6  |  |  t'_b ← bisection neighbour of t'
7  |  |  BisectTriangle(t')
8  |  |  BisectTriangle(t'_b)
9  else
10 |  add t to the sequence of triangles visited by RefineTriangle
11 |  RefineTriangle(t_b)
```

Maubach's method has linear running time with respect to the number of triangles in the final refined mesh. This follows from the bound on the recursion depth of `RefineTriangle` that only depends on the maximum level of refinement in T, and since Maubach's method is based on bisection refinement.

1.2 Triangulation Diameter

A key attribute of triangulations relevant to scientific computing applications is the triangulation diameter h, as the numerical error of computations performed on triangulations can often be estimated in terms of h. The triangulation diameter of T is defined to be the maximum distance between adjacent nodes in the mesh; two nodes are adjacent if they form an edge of a $t \in T$.

In this report the coarse triangulation T_0 is defined to be a uniform triangulated grid, where all $t \in T_0$ are identical regular right-angled triangles with vertices coinciding with the nodes of a square grid with spacing δ. The bisection edge length ℓ_b is given by

$$\ell_b(\delta, l) = \sqrt{2}^{(1-l)}\delta, \tag{1}$$

for $t \in T$ with $l = L(t)$. Equation (1) follows from the Pythagorean theorem and the definition of bisection refinement. It follows that the triangulation diameter of T generated using Maubach's method is given by $h = \sqrt{2}\,\delta$, since $L(t) = 0$ for all $t \in T_0$.

2 Mesh Data Structures

Maubach's method generates an unstructured mesh and hence data structures must be defined with particular fields to store the geometric and topological attributes of the mesh. Many types of mesh data structure exist: some minimise the memory required to store the mesh; others minimise the speed of performing certain operations on the mesh. Ultimately, the choice of a particular mesh data structure is highly context dependent. Refer to Hjelle & Dæhlen [2006] for a general discussion of data structures for triangulations.

It can be seen from Algorithms 1 and 2 that the following mesh data must be stored:

- nodes (which form the triangle vertices)
- references to triangle vertices
- references to triangle neighbours
- refinement levels.

The orientation of the arrangement of triangle vertices is also required to be stored for updating references to triangle neighbours, the discussion of which is postponed until Section 5. The orientation data is also used for point location, which is discussed in Section 7.

For each call to `RefineTriangle`, node and triangle data must be found, new nodes and triangles created, and some existing triangles must have their data updated. Therefore three operations are required to be performed on the mesh data structures: find entries; add new entries; and update existing entries.

A natural choice of mesh data structure is an array, as node and triangle data can be referenced using their unique array indices, and looking up a particular entry in an array is very fast. However, searching and adding entries to an array are expensive operations, especially if the array is very large. Also note that rectangular arrays with entries of the same type (integers, reals, ...) are more computationally efficient to store and access, compared with non-rectangular (ragged) arrays with entries of mixed type.

These observations motivate the choice of mesh data structures employed in the implementation of Maubach's method discussed in this report.

2.1 Attributes

Let $\mathcal{N}$ be the data structure for nodes, where $\mathcal{N}$ is an array with entries $\mathcal{N}[i] = \mathbf{x}_i \in \Omega$. The vertices of $t \in T$ are denoted by $V(t)$ and specified by references to entries in $\mathcal{N}$; for example, if t has the vertices $(\mathbf{x}_{95}, \mathbf{x}_{154}, \mathbf{x}_3)$ then $V(t) = (95, 154, 3)$. For the general case let $V(t) = (v_1, v_2, v_3)$ where $v_i \in \mathbb{N}$ are references to entries in $\mathcal{N}$. The v_i are ordered according to Maubach's method; refer to Algorithm 1.

Let $\mathcal{T}$ be the data structure for triangles, where $\mathcal{T}$ is an array with entries $\mathcal{T}[i] \in \mathbb{Z}^8$ that contain data associated with the ith triangle in T. The triangle neighbours of $t \in T$ are denoted by $N(t) = (n_1, n_2, n_3)$ where $n_i \in \mathbb{Z}$. If $n_i > 0$ then n_i references a triangle in

$\mathcal{T}$, otherwise if $n_i < 0$ then n_i references a null boundary triangle.[3] The n_i are ordered to facilitate the identification of neighbours without any computations:

- n_1 references the bisection neighbour
- neighbours are arranged anticlockwise about t
- if $n_i < 0$ then t has an edge on the boundary of Ω that would be shared with n_i, if it was not a reference to a null boundary triangle.

Let $O(t) \in \{0, 1\}$ represent the orientation of the arrangement of the vertices of $t \in T$, where $O(t) = 0$ indicates the vertices are arranged clockwise and $O(t) = 1$ indicates the vertices have an anticlockwise arrangement. Further discussion of $O(t)$ is postponed until Section 5.

The ith entry of $\mathcal{T}$ is given by

$$\mathcal{T}[i] = (v_{i1}, v_{i2}, v_{i3}, n_{i1}, n_{i2}, n_{i3}, o_i, l_i),$$

where $o_i = O(t_i)$ and $l_i = L(t_i)$. Let $\mathcal{T}[i, j]$ refer to the jth entry in $\mathcal{T}[i]$ for $1 \leqslant j \leqslant 8$, and $\mathcal{T}[i, j : k]$ refer to the jth to kth consecutive entries of $\mathcal{T}[i]$ for $1 \leqslant j < k \leqslant 8$. It follows that

$$\mathcal{T}[i, 1 : 3] = V(t_i),$$

$$\mathcal{T}[i, 4 : 6] = N(t_i),$$

$$\mathcal{T}[i, 7] = O(t_i),$$

$$\mathcal{T}[i, 8] = L(t_i).$$

The attributes of $\mathcal{N}$ and $\mathcal{T}$ facilitate a computationally efficient implementation of Maubach's method:

- $\mathcal{N}$ and $\mathcal{T}$ are rectangular arrays each with entries of the same type
- the order of the entries in $\mathcal{T}[i]$ enables references to triangle vertices and neighbours to be identified without any computations
- the compact structure of $\mathcal{T}[i]$ minimises the need to append triangle data to multiple data structures.

In fact, it will be shown in Section 2.2 that appending data to $\mathcal{N}$ and $\mathcal{T}$ can almost be eliminated.

[3] The case $n_i = 0$ will be discussed in Section 2.2.

2.2 Memory Allocation

The triangulation T is initialised to T_0 and stored in $\mathcal{N}$ and $\mathcal{T}$, then calls to `RefineTriangle` locally refine T resulting in new nodes and triangles being created. One option is to append new node and triangle data to $\mathcal{N}$ and $\mathcal{T}$; however this is an expensive operation. A more efficient alternative is to estimate the sizes of $\mathcal{N}$ and $\mathcal{T}$ necessary to store the final refined T, and pad out $\mathcal{N}$ and $\mathcal{T}$ with the required amount of null data after initialisation. These estimates should be upper bounds as this will eliminate the requirement to append data to $\mathcal{N}$ and $\mathcal{T}$; the excess padding can be removed once the refinement of T is complete.

Maubach's method refines T according to a given local condition, which in this report is assumed to depend on a function that specifies the target bisection edge length b_t for each $t \in T$.[4] To estimate the number of triangles in the final refined T, $L(t)$ will be estimated for each $t \in T_0$ using b_t. Then $\sum_{t \in T_0} 2^{L(t)}$ triangles would be created if each $t \in T_0$ and all their descendants were bisected $L(t)$ times. The inverse bisection edge length $\ell_b^{-1}(\delta, b_t)$ is defined to return the minimum $L(t)$ such that $\ell_b(\delta, L(t)) \leqslant b_t$ and is given by[5]

$$\ell_b^{-1}(\delta, b_t) = \left\lceil 2 \log_2\left(\sqrt{2}\,\delta / b_t\right) \right\rceil,$$

which follows from Equation (1).

The number of triangles in the final refined T is denoted by N_t^{f} and can be estimated in terms of ℓ_b^{-1}:

$$N_t^{\mathrm{f}} \approx \sum_{t \in T_0} 2^{\ell_b^{-1}(\delta, b_t)}. \tag{2}$$

It can be shown that the number of nodes and triangles in a triangulation, and hence the number of entries in $\mathcal{N}$ and $\mathcal{T}$, are related via [Hjelle & Dæhlen 2006, Lemma 1.1]

$$|\mathcal{T}| \sim 2\,|\mathcal{N}| \text{ as } |\mathcal{N}| \to \infty. \tag{3}$$

Hence the number of nodes in the final refined T will be estimated using

$$N_{\mathbf{x}}^{\mathrm{f}} \approx \left\lfloor \frac{1}{2} N_t^{\mathrm{f}} \right\rfloor, \tag{4}$$

with N_t^{f} given by Equation (2).

Since $\mathcal{N}[i] \in \mathbb{R}^2$ and $\mathcal{T}[i] \in \mathbb{Z}^8$, the arrays of null data used to pad out $\mathcal{N}$ and $\mathcal{T}$ are defined to be

$$\mathcal{P}_{\mathcal{N}} = ((0.0, 0.0), \dots),$$

$$\mathcal{P}_{\mathcal{T}} = ((0, 0, 0, 0, 0, 0, 0, 0), \dots).$$

At initialisation of Maubach's method, $T \leftarrow T_0$, the mesh data is stored in $\mathcal{N}$ and $\mathcal{T}$, and then

$$\mathcal{N} \leftarrow \Big(\mathcal{N}[1], \mathcal{N}[2], \dots, \mathcal{N}[N_{\mathbf{x}}], \mathcal{P}_{\mathcal{N}}[1], \dots, \mathcal{P}_{\mathcal{N}}[N_{\mathbf{x}}^{\mathrm{f}} - N_{\mathbf{x}}]\Big), \tag{5}$$

$$\mathcal{T} \leftarrow \Big(\mathcal{T}[1], \mathcal{T}[2], \dots, \mathcal{T}[N_t], \mathcal{P}_{\mathcal{T}}[1], \dots, \mathcal{P}_{\mathcal{T}}[N_t^{\mathrm{f}} - N_t]\Big), \tag{6}$$

[4] The refinement condition is discussed in Section 3.

[5] Note that ℓ_b^{-1} is not a true mathematical inverse.

where N_t^{f} and $N_{\mathbf{x}}^{\mathrm{f}}$ are given by Equations (2) and (4), respectively, and $N_{\mathbf{x}}$ is the number of nodes and N_t the number of triangles in T.

Equations (2) and (4) represent informal upper bounds on N_t^{f} and $N_{\mathbf{x}}^{\mathrm{f}}$, and therefore $\mathcal{N}$ and $\mathcal{T}$ as defined by Equations (5) and (6) may have insufficient padding to store the final refined T. Consequently the residual padding must be checked before each iteration of Maubach's method. `RefineTriangle` is a recursive algorithm and hence the number of nodes and triangles created per call cannot be known in advance, and for this reason the residual padding must be checked before bisection is performed by `BisectTriangle`. Observe from Algorithm 2 that `BisectTriangle` only bisects compatibly divisible triangles and their bisection neighbours, creating one node and two triangles per call.[6] Therefore if

$$|\mathcal{N}| \geqslant N_{\mathbf{x}} + 1, \tag{7}$$

and

$$|\mathcal{T}| \geqslant N_t + 2, \tag{8}$$

then there is sufficient padding in $\mathcal{N}$ and $\mathcal{T}$ for at least one call to `BisectTriangle`. In fact, it is now shown that Equation (7) implies Equation (8).

Let $M_{\mathcal{T}} \geqslant 2$ be a given number of additional null triangles that, if required, can be used to pad out $\mathcal{T}$. Motivated by Equation (4), $\lfloor M_{\mathcal{T}}/2 \rfloor$ null nodes will be used to pad out $\mathcal{N}$ as required. Accordingly, $|\mathcal{P}_{\mathcal{T}}| \geqslant 2\,|\mathcal{P}_{\mathcal{N}}|$ for all iterations of Maubach's method. It follows that if Equation (7) is true, then for all iterations of Maubach's method,

$$\begin{aligned} N_t + 2 &\leqslant |\mathcal{T}| - 2\,|\mathcal{P}_{\mathcal{N}}| + 2, \\ &= |\mathcal{T}| - 2\,(|\mathcal{N}| - N_{\mathbf{x}}) + 2, \\ &= |\mathcal{T}| - 2\,|\mathcal{N}| + 2\,(N_{\mathbf{x}} + 1), \\ &\leqslant |\mathcal{T}|, \end{aligned}$$

and therefore Equation (8) is also true.

The preceding discussion motivates the definition of `CheckMemory` presented in Algorithm 3. Calling `CheckMemory` before bisection is performed by `BisectTriangle` guarantees there will be sufficient null data in $\mathcal{N}$ and $\mathcal{T}$ to store new nodes and triangles.

Algorithm 3: `CheckMemory`

Result: $\mathcal{N}$ and $\mathcal{T}$ contain enough padding to call `BisectTriangle` on a triangle and its compatibly divisible neighbour

```
1 if |𝒩| < N_x + 1 then
2     𝒩 ← (𝒩[1], 𝒩[2], ..., 𝒩[N_x], 𝒫_𝒩[1], ..., 𝒫_𝒩[ ⌊M_𝒯/2⌋ ])
3     𝒯 ← (𝒯[1], 𝒯[2], ..., 𝒯[N_t], 𝒫_𝒯[1], ..., 𝒫_𝒯[M_𝒯])
```

[6] Bisecting a triangle that has its bisection edge on the boundary of Ω creates one node and one triangle per call to `BisectTriangle`; this issue will be considered in Section 5.

3 Local Refinement Condition

The aim of this section is to derive a local refinement condition that returns `True` if `RefineTriangle`(t) must be called, or `False` otherwise; Maubach's method will terminate when this condition returns `False` for all $t \in T$. This local refinement condition will be derived such that the distance between neighbouring nodes in the final refined T will be bounded by a specified target, with the goal of controlling the numerical error of computations performed on T, while curtailing N_t.

First, the local triangulation diameter is defined. The neighbourhood $\mathscr{N}(\mathbf{x})$ of a node $\mathbf{x}$ in T is given by

$$\mathscr{N}(\mathbf{x}) = \{\mathbf{y} \in \Omega \mid \text{there exists a } t \in T \text{ such that } \mathbf{x} \in t \text{ and } \mathbf{y} \in t\}, \tag{9}$$

where the notation $\mathbf{z} \in t$ is to be interpreted as "$\mathbf{z}$ is a vertex of t". The local triangulation diameter $h(\mathbf{x})$ is defined in terms of $\mathscr{N}(\mathbf{x})$:

$$h(\mathbf{x}) = \max_{\mathbf{y} \in \mathscr{N}(\mathbf{x})} \|\mathbf{x} - \mathbf{y}\|. \tag{10}$$

Let $\mathscr{V}(T)$ represent the set of nodes that comprise T, that is,

$$\mathscr{V}(T) = \{\mathbf{x} \in \Omega \mid \text{there exists a } t \in T \text{ such that } \mathbf{x} \in t\}.$$

Equation (10) provides a more general definition of the triangulation diameter h than that given in Section 1.2:

$$h = \max_{\mathbf{x} \in \mathscr{V}(T)} h(\mathbf{x}). \tag{11}$$

Let $\overline{h}(\mathbf{x})$ be the specified target local triangulation diameter for all $\mathbf{x} \in \mathscr{V}(T)$. Since the local refinement condition will be defined in terms of triangles, define $\overline{h}(t)$ via

$$\overline{h}(t) = \min_{\mathbf{x} \in t} \overline{h}(\mathbf{x}), \tag{12}$$

for all $t \in T$. Note that $\overline{h}(t)$ is equivalent to b_t discussed in Section 2.2.

The local refinement condition is given by

$$\ell_b(\delta, L(t)) > \overline{h}(t), \tag{13}$$

where δ is the grid spacing of T_0, and recall that Maubach's method will terminate when Equation (13) returns `False` for all $t \in T$. Let T denote a triangulation generated using Maubach's method. Equation (13) ensures T will obey

$$h(\mathbf{x}) \leqslant \overline{h}(\mathbf{x}) \text{ for all } \mathbf{x} \in \mathscr{V}(T). \tag{14}$$

To verify Equation (14), first let

$$\mathscr{T}(\mathbf{x}) = \{t \in T \mid \mathbf{x} \in t\},$$

and choose any $\mathbf{x} \in \mathscr{V}(T)$. Then

$$\begin{aligned} h(\mathbf{x}) &= \max_{\mathbf{y} \in \mathscr{N}(\mathbf{x})} \|\mathbf{x} - \mathbf{y}\|, \\ &\leqslant \max_{t \in \mathscr{T}(\mathbf{x})} \ell_b(\delta, L(t)) \text{ (property of local bisection refinement)}, \\ &\leqslant \max_{t \in \mathscr{T}(\mathbf{x})} \overline{h}(t) \text{ (by the negation of Equation (13))}, \\ &= \max_{t \in \mathscr{T}(\mathbf{x})} \min_{\mathbf{y} \in t} \overline{h}(\mathbf{y}) \text{ (by Equation (12))}, \\ &\leqslant \overline{h}(\mathbf{x}) \text{ (as } \mathbf{x} \in t \text{ for all } t \in \mathscr{T}(\mathbf{x})). \end{aligned}$$

The given target triangulation diameter $\overline{h}$ is analogously defined to h in Equation (11). Therefore Equation (13) enables h of the final refined T to be controlled using $\overline{h}$, since Equation (14) implies $h \leqslant \overline{h}$, thereby controlling the numerical error of computations performed on T.

4 Coarse Triangulation

Maubach's method begins with a coarse triangulation T_0 of Ω. To simplify the discussion Ω is defined to be $[0,1]^2$; non-square domains and domains with holes are considered in Appendix A. The aim of this section is to present algorithms for constructing T_0.

Maubach's method can simply be initialised using a T_0 containing two triangles that are compatibly divisible. However in this report T_0 is chosen to be a uniform triangulated grid, where all $t \in T_0$ are identical regular right-angled triangles with vertices coinciding with the nodes of a square grid with spacing δ. In Section 7 this choice will be shown to facilitate efficient point location. The assignment of indices to nodes and triangles in T_0 is shown in Figure 2.

The specified maximum grid spacing $\overline{\delta}$ for T is related to δ via

$$\delta = \left\lceil \overline{\delta}^{-1} \right\rceil^{-1}. \tag{15}$$

This definition of δ implies that $h \leqslant \overline{h}$, which is consistent with Equation (14). Furthermore, the number of triangles in T_0 that have an edge on any one side of $[0,1]^2$ is given by

$$N_\mathrm{e} = \left\lceil \overline{\delta}^{-1} \right\rceil.$$

It follows that there are $(N_\mathrm{e}+1)^2$ nodes and $2N_\mathrm{e}^2$ triangles in T_0.

4.1 Construction

The Cartesian coordinates of the nodes that constitute the vertices of the triangles in T_0 are generated by `CoarseNodes` (Algorithm 4) and stored in $\mathcal{N}$. The order of the nodes in $\mathcal{N}$ is shown in Figure 2.

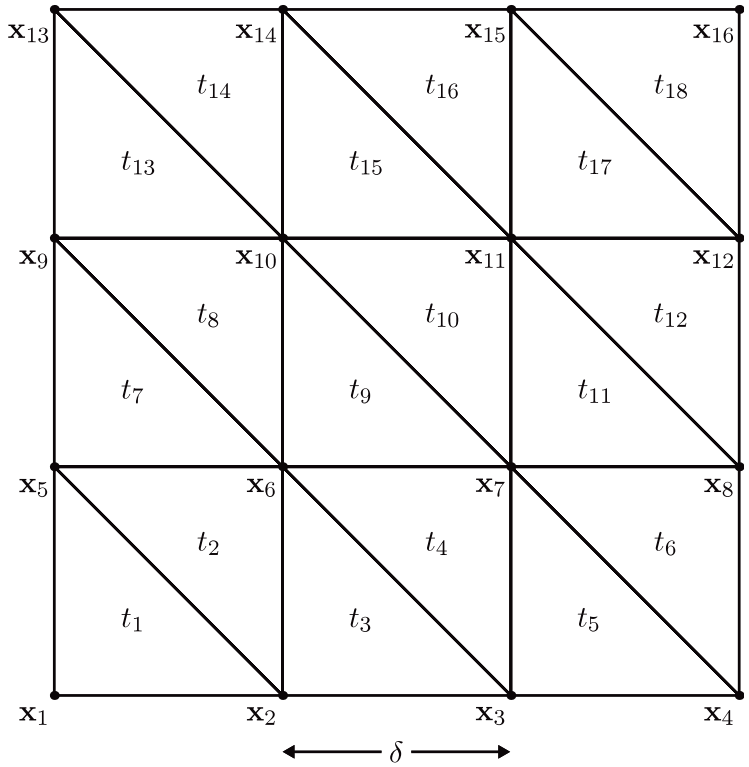


Figure 2: *A coarse triangulation showing the assignment of indices to nodes and triangles, and the grid spacing δ.*

Algorithm 4: `CoarseNodes`

Input: $N_\mathrm{e} \in \mathbb{N}$
Output: $\mathcal{N}$

$\delta \leftarrow N_\mathrm{e}^{-1}$
$\mathcal{N} \leftarrow \left(\mathcal{P}_\mathcal{N}[1], \ldots, \mathcal{P}_\mathcal{N}[(N_\mathrm{e}+1)^2]\right)$
$k \leftarrow 1$
foreach $i \in (0, 1, \ldots, N_\mathrm{e})$ **do**
  foreach $j \in (0, 1, \ldots, N_\mathrm{e})$ **do**
    $\mathcal{N}[k] \leftarrow \delta\,(j, i)$
    $k \leftarrow k + 1$
return $\mathcal{N}$

The vertices of the triangles in T_0 are stored in $\mathcal{T}_0^\mathrm{V}$ as references to their unique entries in $\mathcal{N}$. References to the vertices of the ith triangle are stored in $\mathcal{T}_0^\mathrm{V}[i]$: the order of the $\mathcal{T}_0^\mathrm{V}[i]$ agrees with the assignment of indices to triangles shown in Figure 2. Since $L(t) = 0$ for all $t \in T_0$, the order of the three references in each $\mathcal{T}_0^\mathrm{V}[i]$ corresponds to an anticlockwise arrangement of the vertices such that the first and last vertices form the bisection edge; see Algorithm 1. The data structure $\mathcal{T}_0^\mathrm{V}$ is constructed by `CoarseTriangleVertices` as

per Algorithm 5. Note that $\mathcal{T}_0^{\mathrm{V}}$ only contains references to vertices and therefore $\mathcal{T}_0^{\mathrm{V}}$ is initialised using the null data structure given by $\mathcal{P}_{\mathcal{T}}^0 = ((0,0,0),\dots)$.

Algorithm 5: `CoarseTriangleVertices`

Input: $N_{\mathrm{e}} \in \mathbb{N}$
Output: $\mathcal{T}_0^{\mathrm{V}}$

$n \leftarrow N_{\mathrm{e}} + 1$
$\mathcal{T}_0^{\mathrm{V}} \leftarrow \left(\mathcal{P}_{\mathcal{T}}^0[1], \dots, \mathcal{P}_{\mathcal{T}}^0[2N_{\mathrm{e}}^2]\right)$
$k \leftarrow 1$
foreach $i \in (0, 1, \dots, N_{\mathrm{e}} - 1)$ **do**
    foreach $j \in (0, 1, \dots, N_{\mathrm{e}} - 1)$ **do**
        $\mathcal{T}_0^{\mathrm{V}}[k,1] \leftarrow j + 1 + n\,(i+1)$
        $\mathcal{T}_0^{\mathrm{V}}[k,2] \leftarrow j + 1 + ni$
        $\mathcal{T}_0^{\mathrm{V}}[k,3] \leftarrow j + 2 + ni$
        $k \leftarrow k + 1$
        $\mathcal{T}_0^{\mathrm{V}}[k,1] \leftarrow j + 2 + ni$
        $\mathcal{T}_0^{\mathrm{V}}[k,2] \leftarrow j + 2 + n\,(i+1)$
        $\mathcal{T}_0^{\mathrm{V}}[k,3] \leftarrow j + 1 + n\,(i+1)$
        $k \leftarrow k + 1$
return $\mathcal{T}_0^{\mathrm{V}}$

The neighbours of the ith triangle in T_0 are stored in $\mathcal{T}_0^{\mathrm{N}}[i]$ as references to their unique triangle indices; the assignment of indices to triangles is shown in Figure 2. The order of the three references in each $\mathcal{T}_0^{\mathrm{N}}[i]$ coincides with the attributes of $\mathcal{T}$ for storing triangle neighbours; see Section 2.1. The data structure $\mathcal{T}_0^{\mathrm{N}}$ is constructed by `CoarseTriangleNeighbours` in accordance with Algorithm 6, where the index $\beta < 0$ denotes a reference to a null boundary triangle. Observe from Figure 2 that triangles with an odd index may have a boundary edge on the left and bottom segments of the boundary of T_0, whereas triangles with an even index may have a boundary edge on the right and top segments of the boundary of T_0; only the first and last triangles have two boundary edges.

Algorithm 6: `CoarseTriangleNeighbours`

Input: $N_\mathrm{e} \in \mathbb{N}$
Output: $\mathcal{T}_0^\mathrm{N}$

$n \leftarrow 2N_\mathrm{e}^2$ `// number of triangles in` T_0
$s \leftarrow 2N_\mathrm{e}$ `// number of triangles in a strip`
$l \leftarrow 1$ `// left boundary indicator`
$r \leftarrow 1$ `// right boundary indicator`
$k \leftarrow n - s + 1$
$\mathcal{T}_0^\mathrm{N} \leftarrow \left(\mathcal{P}_\mathcal{T}^0[1], \ldots, \mathcal{P}_\mathcal{T}^0[n]\right)$
$\mathcal{T}_0^\mathrm{N}[1] \leftarrow (2, \beta, \beta)$
$\mathcal{T}_0^\mathrm{N}[n] \leftarrow (n-1, \beta, \beta)$
foreach $i \in (2, 3, \ldots, n-1)$ **do**
    if i is odd **then**
        if $i = ls + 1$ **then**
            $l \leftarrow l + 1$
            $\mathcal{T}_0^\mathrm{N}[i] \leftarrow (i+1, \beta, i-s+1)$
        else if $i > s$ **then**
            $\mathcal{T}_0^\mathrm{N}[i] \leftarrow (i+1, i-1, i-s+1)$
        else
            $\mathcal{T}_0^\mathrm{N}[i] \leftarrow (i+1, i-1, \beta)$
    else
        if $i = rs$ **then**
            $r \leftarrow r + 1$
            $\mathcal{T}_0^\mathrm{N}[i] \leftarrow (i-1, \beta, i+s-1)$
        else if $i < k$ **then**
            $\mathcal{T}_0^\mathrm{N}[i] \leftarrow (i-1, i+1, i+s-1)$
        else
            $\mathcal{T}_0^\mathrm{N}[i] \leftarrow (i-1, i+1, \beta)$
return $\mathcal{T}_0^\mathrm{N}$

4.2 Mesh Initialisation

The results of the previous sections can now be employed to initialise T. First T_0 is constructed in accordance with Algorithms 4 to 6 and stored in the data structures $\mathcal{N}$ and $\mathcal{T}$; the variables $N_{\mathbf{x}}$ and N_t are also stored. Equations (2) and (4) are then used to estimate the number of triangles in the final refined T, and the amount of null data required to store T is appended to $\mathcal{N}$ and $\mathcal{T}$. This procedure is executed by `InitialiseMesh` (Algorithm 7), which has $\mathcal{O}(N_t)$ running time.

Algorithm 7: `InitialiseMesh`

Input: $\overline{\delta} > 0$ and $\overline{h}(\,\cdot\,)$ as per Equation (16)
Result: T is initialised and stored in $\mathcal{N}$ and $\mathcal{T}$; $N_{\mathbf{x}}$ and N_t are also stored

1 $N_{\mathrm{e}} \leftarrow \lceil 1/\overline{\delta} \rceil$
2 $N_{\mathbf{x}} \leftarrow (N_{\mathrm{e}} + 1)^2$
3 $N_t \leftarrow 2N_{\mathrm{e}}^2$
4 $\mathcal{N} \leftarrow$ `CoarseNodes`(N_{e})
5 $\mathcal{T}_0^{\mathrm{V}} \leftarrow$ `CoarseTriangleVertices`(N_{e})
6 $\mathcal{T}_0^{\mathrm{N}} \leftarrow$ `CoarseTriangleNeighbours`(N_{e})
7 $\mathcal{T} \leftarrow (\mathcal{P}_{\mathcal{T}}[1], \ldots, \mathcal{P}_{\mathcal{T}}[N_t])$
8 **foreach** $i \in (1, 2, \ldots, N_t)$ **do**
9 $\quad \mathcal{T}[i] \leftarrow \big(\mathcal{T}_0^{\mathrm{V}}[i,1], \mathcal{T}_0^{\mathrm{V}}[i,2], \mathcal{T}_0^{\mathrm{V}}[i,3], \mathcal{T}_0^{\mathrm{N}}[i,1], \mathcal{T}_0^{\mathrm{N}}[i,2], \mathcal{T}_0^{\mathrm{N}}[i,3], 1, 0\big)$
10 $\delta \leftarrow 1/N_{\mathrm{e}}$
11 $N_t^{\mathrm{f}} \leftarrow \sum_{i=1}^{N_t} 2^{\ell_b^{-1}(\delta, \overline{h}(i))}$
12 $N_{\mathbf{x}}^{\mathrm{f}} \leftarrow \lfloor N_t^{\mathrm{f}}/2 \rfloor$
13 $\mathcal{N} \leftarrow \big(\mathcal{N}[1], \mathcal{N}[2], \ldots, \mathcal{N}[N_{\mathbf{x}}], \mathcal{P}_{\mathcal{N}}[1], \ldots, \mathcal{P}_{\mathcal{N}}[N_{\mathbf{x}}^{\mathrm{f}} - N_{\mathbf{x}}]\big)$
14 $\mathcal{T} \leftarrow \big(\mathcal{T}[1], \mathcal{T}[2], \ldots, \mathcal{T}[N_t], \mathcal{P}_{\mathcal{T}}[1], \ldots, \mathcal{P}_{\mathcal{T}}[N_t^{\mathrm{f}} - N_t]\big)$

Algorithms 5, 6 and the initialisation of $\mathcal{T}$ on line 9 of Algorithm 7 establish the attributes of $\mathcal{T}$ discussed in Section 2.1. Also note that the target local triangulation diameter function, given by Equation (12), is modified to take a triangle index as an argument for use on line 11 of Algorithm 7:

$$\overline{h}(i) = \min_{j \in \mathcal{T}[i,1:3]} \overline{h}(\mathcal{N}[j]), \tag{16}$$

for $i \in \{1, 2, \ldots, N_t\}$.

5 Mesh Refinement

This section concludes the implementation of Maubach's method. Algorithms for simultaneously bisecting compatibly divisible triangles and updating their neighbours are examined; these are the most important algorithms of the implementation as their behaviour is critically linked to Maubach's method and the data structures $\mathcal{N}$ and $\mathcal{T}$. The top-level mesh refinement algorithm is also presented, together with a demonstration of its computational performance.

5.1 Bisecting Triangles

`BisectTriangle` bisects a given triangle $t_i \in T$ and creates the two descendants. A high-level description of `BisectTriangle` is given in Algorithm 1. The objective of this section is to provide a detailed implementation of an algorithm that simultaneously bisects two compatibly divisible triangles in accordance with Maubach's method.

It will be shown in Section 7 that point location can be achieved using a constant running-time algorithm without storing descendants. Therefore, information regarding t_i is not retained after its bisection and the index i is reused to reference one of the descendants of t_i. The other descendant of t_i is given the index $\bar{i} = N_t + 1$, where N_t represents the state of N_t immediately prior to bisecting t_i.

Maubach's method stipulates that the vertices of descendants be ordered to enable the descendants' bisection edges to be subsequently identified without any computations. If t_i is the triangle to be bisected, this order depends on the value of $k(t_i)$, where

$$k(t_i) = 2 - (L(t_i) \mod 2) \in \{1, 2\}.$$

Let $V(t_i) = (a, b, c)$. If $k(t_i) = 1$, then a and b reference the nodes that form the bisection edge, otherwise a and c reference the nodes that form the bisection edge.

Consider bisecting t_i for the case $k(t_i) = 1$ and $O(t_i) = 0$, which corresponds to a clockwise arrangement of the vertices. Then by Maubach's method, the vertices of the descendants t_i and $t_{\bar{i}}$ will satisfy

$$V(t_i) = (a, d, c), \tag{17}$$

$$V(t_{\bar{i}}) = (b, d, c), \tag{18}$$

where the new node with index d is stored in $\mathcal{N}[N_{\mathbf{x}} + 1]$, that is, $d = N_{\mathbf{x}} + 1$. The configuration of the descendants t_i and $t_{\bar{i}}$ is shown in Figure 3 for the case being considered. Observe from Figure 3 and Equations (17) and (18) that the descendants satisfy $O(t_i) = 0$ and $O(t_{\bar{i}}) = 1$.

This example reveals the dependence of the configuration of the descendants of t_i on $k(t_i)$ and $O(t_i)$. The configuration of descendants must be known to update triangle neighbours without any computations, which is discussed in Section 5.2. Since $k(t_i)$ and $O(t_i)$ can each take two values, there are four configurations of the descendants. The remaining three configurations can be extrapolated from Figure 3 by ordering the vertices according to Maubach's method.

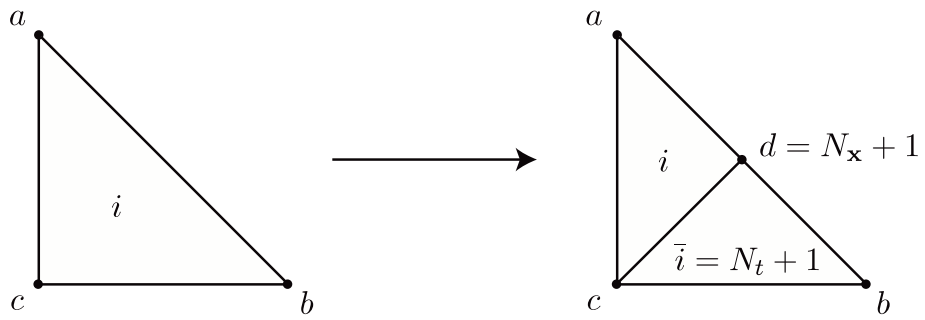


Figure 3: *The configuration of the descendants of t_i for the case $k(t_i) = 1$ and $O(t_i) = 0$. The references to the descendants are i and $\bar{i}$, and a, b, c and d denote references to vertices. Note that $N_{\mathbf{x}}$ and N_t represent their states immediately prior to bisecting t_i.*

Let i be the index of a triangle to be bisected and i_b the index of its bisection neighbour. If t_{i_b} is not a null boundary triangle and t_i and t_{i_b} are compatibly divisible, then `BisectTriangles`(i) (Algorithm 8) simultaneously bisects t_i and t_{i_b}, creating their descendants in accordance with Maubach's method, and updates their neighbours.

The bisection of a triangle with its bisection edge on the boundary of Ω is undertaken by `BisectBoundaryTriangle` (Algorithm 19 in Appendix B). A compatibly divisible triangle with a non-bisection edge on the boundary of Ω is bisected using `BisectTriangles`.

Note that for the algorithms in Section 5 and Appendix B, the symbols $N_{\mathbf{x}}$ and N_t appearing in the "Input" and "Result" fields represent the states of $N_{\mathbf{x}}$ and N_t immediately prior to calling the routines.

Algorithm 8: `BisectTriangles`

Input: $i \in \{1, 2, \ldots, N_t\}$ such that $\mathcal{T}[i, 4] > 0$ and $\mathcal{T}[i, 8] = \mathcal{T}[i_b, 8]$
Result: $\mathcal{T}[i]$ and $\mathcal{T}[i_b]$ are updated; $\mathcal{N}[N_\mathbf{x} + 1]$, $\mathcal{T}[N_t + 1]$ and $\mathcal{T}[N_t + 2]$ are created

`CheckMemory`( )

$i_b \leftarrow \mathcal{T}[i, 4]$
$V \leftarrow \mathcal{T}[i, 1:3]$
$V_b \leftarrow \mathcal{T}[i_b, 1:3]$

$N_\mathbf{x} \leftarrow N_\mathbf{x} + 1$
$k(i) \leftarrow 2 - (\mathcal{T}[i, 8] \mod 2)$
$\mathcal{N}[N_\mathbf{x}] \leftarrow 0.5\,(\mathcal{N}[V[1]] + \mathcal{N}[V[k+1]])$

$N_t \leftarrow N_t + 2$
$O \leftarrow (\mathcal{T}[i, 7] + 1) \mod 2$
$O_b \leftarrow (\mathcal{T}[i_b, 7] + 1) \mod 2$
$\mathcal{T}[i, 8] \leftarrow \mathcal{T}[i, 8] + 1$
$\mathcal{T}[i_b, 8] \leftarrow \mathcal{T}[i_b, 8] + 1$

if $k(i) = 1$ **then**
  $\mathcal{T}[i, 1] \leftarrow V[1]$
  $\mathcal{T}[i, 2] \leftarrow N_\mathbf{x}$
  $\mathcal{T}[i, 3] \leftarrow V[3]$
  $\mathcal{T}[i_b, 1] \leftarrow V_b[1]$
  $\mathcal{T}[i_b, 2] \leftarrow N_\mathbf{x}$
  $\mathcal{T}[i_b, 3] \leftarrow V_b[3]$
  $\mathcal{T}[N_t - 1] \leftarrow (V[2], N_\mathbf{x}, V[3], 0, 0, 0, O, \mathcal{T}[i, 8])$
  $\mathcal{T}[N_t] \leftarrow (V_b[2], N_\mathbf{x}, V_b[3], 0, 0, 0, O_b, \mathcal{T}[i_b, 8])$
else
  $\mathcal{T}[i, 1] \leftarrow V[1]$
  $\mathcal{T}[i, 2] \leftarrow V[2]$
  $\mathcal{T}[i, 3] \leftarrow N_\mathbf{x}$
  $\mathcal{T}[i_b, 1] \leftarrow V_b[1]$
  $\mathcal{T}[i_b, 2] \leftarrow V_b[2]$
  $\mathcal{T}[i_b, 3] \leftarrow N_\mathbf{x}$
  $\mathcal{T}[N_t - 1] \leftarrow (V[2], V[3], N_\mathbf{x}, 0, 0, 0, \mathcal{T}[i, 7], \mathcal{T}[i, 8])$
  $\mathcal{T}[N_t] \leftarrow (V_b[2], V_b[3], N_\mathbf{x}, 0, 0, 0, \mathcal{T}[i_b, 7], \mathcal{T}[i_b, 8])$

switch $(k(i), \mathcal{T}[i, 7], \mathcal{T}[i_b, 7])$ **do**
  case $(1, 0, 0)$ or $(2, 1, 1)$
    `UpdateNeighbours1`(i)
  case $(1, 1, 0)$ or $(2, 0, 1)$
    `UpdateNeighbours2`(i)
  case $(1, 0, 1)$ or $(2, 1, 0)$
    `UpdateNeighbours3`(i)
  otherwise
    `UpdateNeighbours4`(i)

5.2 Updating Triangle Neighbours

It was shown in Section 5.1 that the configuration of the descendants of t_i can be determined from the values of $k(t_i)$ and $O(t_i)$ prior to bisection. This will now be used in conjunction with the attributes of $\mathcal{T}$ to update the references to neighbours of compatibly divisible triangles post bisection, without any computations.

Consider bisecting compatibly divisible triangles t_i and t_{i_b} for the case $k(t_i) = 1$, $O(t_i) = 0$ and $O(t_{i_b}) = 0$. The attributes of $\mathcal{T}$ stipulate that i_b and i be the first references to the neighbours of t_i and t_{i_b}, respectively. Let the second and third references to the neighbours of t_i and t_{i_b} satisfy

$$N(t_i) = (i_b, a, b), \tag{19}$$

$$N(t_{i_b}) = (i, c, d). \tag{20}$$

The configuration of t_i and t_{i_b} and their neighbours prior to bisection is shown on the left of Figure 4; this follows from Equations (19) and (20) and the attributes of $\mathcal{T}$. The configuration of the descendants of t_i and t_{i_b} and their neighbours is shown on the right of Figure 4; this follows from Maubach's method as per Figure 3. Observe that t_i and t_b are no longer neighbours after bisection; likewise, t_{i_b} and t_d are no longer neighbours. Therefore $N(t_b)$ and $N(t_d)$ must also be updated to reflect these changes; this is performed by calling `ReplaceTriangleNeighbour` (Algorithm 20 in Appendix B). `UpdateNeighbours1`(i), which is called by `BisectTriangles` (Algorithm 8), updates the neighbours of the descendants of t_i and t_{i_b} for the case being considered. An implementation of `UpdateNeighbours1` is presented in Algorithm 9.

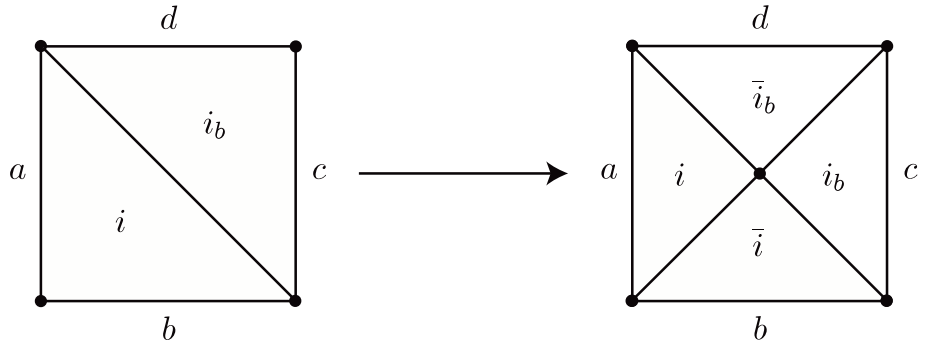


Figure 4: *The configuration of triangle neighbours before and after the bisection of compatibly divisible triangles t_i and t_{i_b} for the case $k(t_i) = 1$, $O(t_i) = 0$ and $O(t_{i_b}) = 0$. Here a, b, c and d represent triangle neighbours.*

This example reveals the dependence of the configuration of neighbours on $k(t_i)$, $O(t_i)$ and $O(t_{i_b})$. Since each of these can take two values, there are potentially eight configurations of neighbours. However it can be shown that only four of these configurations are independent. The remaining three independent configurations can be extrapolated from Figures 3 and 4 by employing Maubach's method and the attributes of $\mathcal{T}$. Implementations of `UpdateNeighbours` for these cases can be found in Algorithms 21 to 23 in Appendix B.

Algorithm 9: `UpdateNeighbours1`

Input: $i \in \{1, 2, \ldots, N_t - 2\}$ such that $(k(i), \mathcal{T}[i, 7], \mathcal{T}[i_b, 7]) = (1, 0, 0)$ or $(2, 1, 1)$
Result: $\mathcal{T}[i, 4:6]$ and $\mathcal{T}[i_b, 4:6]$ are updated; $\mathcal{T}[N_t - 1, 4:6]$ and $\mathcal{T}[N_t, 4:6]$ are created

1 $\bar{i} \leftarrow N_t - 1$
2 $\bar{i}_b \leftarrow N_t$
3 $i_b \leftarrow \mathcal{T}[i, 4]$
4 $N \leftarrow \mathcal{T}[i, 5:6]$
5 $N_b \leftarrow \mathcal{T}[i_b, 5:6]$
6 $\mathcal{T}[i, 4] \leftarrow N[1]$
7 $\mathcal{T}[i, 5] \leftarrow \bar{i}$
8 $\mathcal{T}[i, 6] \leftarrow \bar{i}_b$
9 $\mathcal{T}[i_b, 4] \leftarrow N_b[1]$
10 $\mathcal{T}[i_b, 5] \leftarrow \bar{i}_b$
11 $\mathcal{T}[i_b, 6] \leftarrow \bar{i}$
12 $\mathcal{T}[\bar{i}, 4] \leftarrow N[2]$
13 $\mathcal{T}[\bar{i}, 5] \leftarrow i_b$
14 $\mathcal{T}[\bar{i}, 6] \leftarrow i$
15 $\mathcal{T}[\bar{i}_b, 4] \leftarrow N_b[2]$
16 $\mathcal{T}[\bar{i}_b, 5] \leftarrow i$
17 $\mathcal{T}[\bar{i}_b, 6] \leftarrow i_b$
18 `ReplaceTriangleNeighbour`$(N[2], i, \bar{i})$
19 `ReplaceTriangleNeighbour`$(N_b[2], i_b, \bar{i}_b)$

Consider bisecting a triangle t_i with its bisection edge on the boundary of Ω. The configuration of the descendants of t_i and their neighbours only depends on $k(t_i)$ and $O(t_i)$, as t_i does not have a neighbour on its bisection edge prior to bisection. The neighbours of the descendants of t_i are updated by calling `UpdateBoundaryNeighbours1` or `UpdateBoundaryNeighbours2`, depending on the values of $k(t_i)$ and $O(t_i)$; implementations of these routines can be found in Appendix B.

5.3 Refinement Algorithm

An updated version of `RefineTriangle` that incorporates the algorithms of the previous sections is presented in Algorithm 10. Most of the challenges of implementing Maubach's method originate in the data structures for storing T and the routines for updating triangle neighbours. This is reflected by the similarity of Algorithm 10 and the more abstract version of `RefineTriangle` appearing in Algorithm 2.

`RefineTriangle`(i) calls itself repeatedly, starting with t_i, until a compatibly divisible triangle t_c is found; see Algorithm 10. This generates a sequence of triangles. A stack (last-in-first-out queue) $\mathcal{S}_b$ enables the triangles in this sequence to be bisected in reverse order from t_c to t_i. Therefore compatibility is preserved because only compatibly divisible triangles are bisected; this is performed by Algorithm 11. `RefineMesh` (Algorithm 12) initialises $\mathcal{S}_b$ then calls `RefineTriangle` repeatedly until the local refinement condition (on line 5 of Algorithm 12) returns `False` for all $t \in T$, and then removes the excess null data from $\mathcal{N}$ and $\mathcal{T}$.

Finally, a refined mesh on $[0,1]^2$ is generated using Maubach's method by specifying the maximum grid spacing $\overline{\delta}$ and the target local triangulation diameter $\overline{h}(\,\cdot\,)$, and calling `InitialiseMesh`$\big(\overline{\delta}, \overline{h}(\,\cdot\,)\big)$ followed by `RefineMesh`$\big(\overline{\delta}, \overline{h}(\,\cdot\,)\big)$. Extending this implementation to allow for non-square domains and domains with holes is considered in Appendix A.

Algorithm 10: `RefineTriangle`

Input: $i \in \{1, 2, \ldots, N_t\}$
Result: t_i is compatibly refined

```
1  i_b ← T[i, 4]
2  if i_b < 0 then
3  |   BisectBoundaryTriangle(i)
4  |   BisectTrianglesInStack( )
5  else if T[i, 8] = T[i_b, 8] then
6  |   BisectTriangles(i)
7  |   BisectTrianglesInStack( )
8  else
9  |   push i onto S_b
10 |   RefineTriangle(i_b)
```

Algorithm 11: `BisectTrianglesInStack`

Result: triangles in stack $\mathcal{S}_b$ are compatibly bisected

```
1  while S_b is not empty do
2  |   j ← pop the top element from S_b
3  |   BisectTriangles(j)
```

Algorithm 12: `RefineMesh`

Input: $\overline{\delta} > 0$ and $\overline{h}(\,\cdot\,)$ as per Equation (16)
Result: the final refined T is stored in $\mathcal{N}$ and $\mathcal{T}$

initialise $\mathcal{S}_b$
$\delta \leftarrow 1/\left\lceil 1/\overline{\delta} \right\rceil$ `// see Equation (15)`
$i \leftarrow 1$
while $i \leqslant N_t$ **do**
  while $\ell_b(\delta, \mathcal{T}[i, 8]) > \overline{h}(i)$ **do**
    `RefineTriangle`(i)
  $i \leftarrow i + 1$
$\mathcal{N} \leftarrow \mathcal{N}[1 : N_{\mathbf{x}}]$
$\mathcal{T} \leftarrow \mathcal{T}[1 : N_t]$

5.4 Computational Performance

It was noted in Section 1.1 that Maubach's method has $\mathcal{O}(N_t)$ running time. It will now be demonstrated that $\mathcal{O}(N_t)$ running time can be attained in practice when `InitialiseMesh` and `RefineMesh` are used for mesh generation.

The mesh shown in Figure 1 was used for testing the computational performance of `InitialiseMesh` and `RefineMesh`. The maximum grid spacing was fixed at $\overline{\delta} = 0.05$ and the target local triangulation diameter was chosen to be

$$\overline{h}(\mathbf{x}) = \begin{cases} d, & \|\mathbf{x}\|_1 < 0.4 \\ 1, & \text{otherwise,} \end{cases}$$

where $0 < d < \sqrt{2}\,\overline{\delta}$. Values of d were chosen that resulted in the final refined triangulations containing between 9,560 and 2,113,944 triangles, with a maximum 13 levels of refinement. Average CPU times and a least-squares line of best fit were calculated after running `InitialiseMesh` and `RefineMesh` 10 times for each value of d.[7]

Figure 5 shows the scaled average CPU times $\hat{\tau} = \tau/\tau_{9560}$ versus $\hat{N}_t = N_t/9560$, where τ_{9560} is the average CPU time to generate the refined triangulation with 9,560 triangles. The $\mathcal{O}(N_t)$ running time of `InitialiseMesh` and `RefineMesh` for the triangulations being considered is evident in Figure 5, where the line of best fit has a gradient of approximately 1.01.

[7] 95% confidence intervals were also calculated, but they were too small to be visible on the plot.

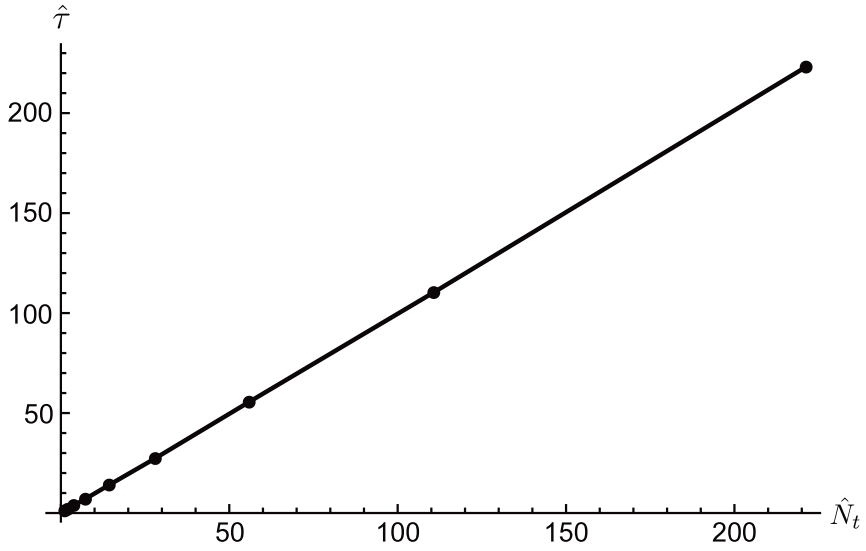


Figure 5: *The scaled average running time of the implementation of Maubach's method documented in this report, versus the scaled number of triangles; a line of best fit is shown with the data points.*

6 Adjacency List

Finding the neighbours of a given node is a common operation on a mesh. The neighbours of a node $\mathbf{x}$ are the nodes that are adjacent to $\mathbf{x}$; two nodes are adjacent if they form an edge of a triangle.[8] An adjacency list $\mathcal{A}$ is an ordered collection of unordered lists, where $\mathcal{A}[i]$ is the set of all node indices that reference the neighbours of the node $\mathbf{x}_i$.

The adjacency list can be constructed during mesh generation in an analogous manner to the initialisation and maintenance of the mesh data for triangle neighbours; see Sections 4.1 and 5.2. Alternatively, the adjacency list can be constructed from an existing mesh; this approach is considered here.

`AdjacencyList` (Algorithm 13) constructs $\mathcal{A}$ by finding all the edges in T. Edges are constructed from triangle vertex references by visiting each $t \in T$ once; this procedure duplicates references to node neighbours as each edge is shared by two triangles.[9] Therefore each list in $\mathcal{A}$ must have capacity for 16 indices, since a node can have at most eight neighbours in a mesh generated using Maubach's method. After finding all the edges in T, the duplicated indices and any excess padding are deleted from each list in $\mathcal{A}$.

[8] The neighbours of $\mathbf{x}$ can also be given by the set $\mathscr{N}(\mathbf{x}) \setminus \mathbf{x}$, where $\mathscr{N}(\mathbf{x})$ is given by Equation (9).
[9] Boundary edges are an exception.

Algorithm 13: `AdjacencyList`

Input: $N_{\mathbf{x}}$ and $\mathcal{T}^{\mathrm{V}}$
Output: $\mathcal{A}$

1 $p_V \leftarrow ((2,3),(1,3),(1,2))$
2 $p_{\mathcal{A}} \leftarrow (1,\dots)$ // $|p_{\mathcal{A}}| = N_{\mathbf{x}}$
3 $\mathcal{A} \leftarrow ((0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),\dots)$ // $|\mathcal{A}| = N_{\mathbf{x}}$
4 **foreach** $i \in \left(1,2,\dots,\left|\mathcal{T}^{\mathrm{V}}\right|\right)$ **do**
5 **foreach** $j \in (1,2,3)$ **do**
6 $n \leftarrow \mathcal{T}^{\mathrm{V}}[i,j]$
7 $n_p \leftarrow p_{\mathcal{A}}[n]$
8 $\mathcal{A}[n,n_p] \leftarrow \mathcal{T}^{\mathrm{V}}[i,p_V[j,1]]$
9 $\mathcal{A}[n,n_p+1] \leftarrow \mathcal{T}^{\mathrm{V}}[i,p_V[j,2]]$
10 $p_{\mathcal{A}}[n] \leftarrow n_p+2$
11 **foreach** $i \in (1,2,\dots,N_{\mathbf{x}})$ **do**
12 $\mathcal{A}[i] \leftarrow$ delete duplicates from $\mathcal{A}[i]$
13 $\mathcal{A}[i] \leftarrow$ delete zero from $\mathcal{A}[i]$
14 **return** $\mathcal{A}$

`AdjacencyList` takes $N_{\mathbf{x}}$ and $\mathcal{T}^{\mathrm{V}}$ as inputs, where $\mathcal{T}^{\mathrm{V}}$ is a data structure containing references to the triangle vertices of T; $\mathcal{T}^{\mathrm{V}}[i] = \mathcal{T}[i,1:3]$ for $i \in \{1,2,\dots,N_t\}$. References to the vertices that form unordered edges of all $t \in T$ are obtained from $p_V = ((2,3),(1,3),(1,2))$ in Algorithm 13, that is, $(\mathcal{T}^{\mathrm{V}}[i,2],\mathcal{T}^{\mathrm{V}}[i,3])$, $(\mathcal{T}^{\mathrm{V}}[i,1],\mathcal{T}^{\mathrm{V}}[i,3])$ and $(\mathcal{T}^{\mathrm{V}}[i,1],\mathcal{T}^{\mathrm{V}}[i,2])$ are references to the unordered edges of t_i. The list $p_{\mathcal{A}}$ in Algorithm 13 enables node neighbours to be added to the lists in $\mathcal{A}$ via an in-place change: the next neighbour of node i to be found is placed in position $p_{\mathcal{A}}[i]$ of $\mathcal{A}[i]$. Most programming languages provide functions for efficiently deleting duplicates from a list, and hence the details of line 12 of Algorithm 13 are omitted.

The implementation of `AdjacencyList` given by Algorithm 13 has the following properties:

- $\mathcal{A}$ is constructed in $\mathcal{O}(N_{\mathbf{x}})$ running time, as the two loop bodies each have $\mathcal{O}(1)$ running time
- $\mathcal{O}(N_{\mathbf{x}})$ memory is required to store $\mathcal{A}$
- references to node neighbours are obtained from $\mathcal{A}$ in fast $\mathcal{O}(1)$ running time.

7 Point Location

Given a query point $\mathbf{y} \in \Omega$, find a $t \in T$ that geometrically contains $\mathbf{y}$. This is known as the point location problem. Point location is a fundamental operation on a mesh and therefore has many applications [Mücke, Saias & Zhu 1999, Soukal, Malková & Kolingerová 2012].

One prominent application of point location is interpolation. Let $q(\mathbf{x})$ be a quantity that is known at each $\mathbf{x} \in \mathscr{V}(T)$, and suppose there is a requirement to calculate $q(\mathbf{y})$ for some $\mathbf{y} \in \Omega$. After using a point location method to find a $t \in T$ that contains $\mathbf{y}$, $q(\mathbf{y})$ can be interpolated from the vertices $\mathbf{x}_i \in t$ using, for example, barycentric co-ordinates:

$$\zeta_1 + \zeta_2 + \zeta_3 = 1,$$

$$\mathbf{y} = \zeta_1 \mathbf{x}_1 + \zeta_2 \mathbf{x}_2 + \zeta_3 \mathbf{x}_3,$$

$$q(\mathbf{y}) = \zeta_1 q(\mathbf{x}_1) + \zeta_2 q(\mathbf{x}_2) + \zeta_3 q(\mathbf{x}_3).$$

Point location is accomplished in two steps:

1. select a $t \in T$ to begin the search
2. traverse T until a $t \in T$ containing the query point $\mathbf{y}$ is found.

A judicious selection of the initial triangle is critical for an efficient implementation of point location. A dedicated data structure can be used to perform the second step; a binary tree is an obvious choice for a mesh generated using Maubach's method. Alternatively, the existing triangle neighbour data can be employed to "walk" from triangle to triangle until a triangle containing $\mathbf{y}$ is found. The two steps of point location are examined in the following two sections.

7.1 Initial Triangle Selection

A technique that subdivides Ω into "buckets" using a uniform grid can be used to select the initial triangle by identifying a bucket containing $\mathbf{y}$ [Asano et al. 1985]. The simplicity of this subdivision enables a bucket containing $\mathbf{y}$ to be found in constant running time, however additional memory is required to store the buckets.

An alternative to bucketing that does not require additional memory begins by randomly choosing a subset of points from $\mathscr{V}(T)$. A point from this subset that minimises the distance to $\mathbf{y}$ is then selected to be the starting point of a straight-line walk to $\mathbf{y}$. The resulting point location algorithm has an $\mathcal{O}(N_{\mathbf{x}}^{1/3})$ expected running time [Mücke, Saias & Zhu 1999].

For a mesh generated using Maubach's method, the initial triangle can be selected in constant running time without requiring additional memory. This follows from three factors:

- Maubach's method uses bisection refinement and therefore all descendants of a coarse triangle are geometrically contained within that triangle
- data regarding the history of descendants is not stored and the index of a parent triangle is reused to reference one of its descendants
- Algorithm 7 generates a uniform triangulated grid T_0 to initialise Maubach's method.

Hence if i is the index of a coarse triangle in T_0 that contains $\mathbf{y}$, then the vertices of $t_i \in T$ are at most a distance of $\overline{h}$ from $\mathbf{y}$.

Fast initial triangle selection for point location is the key reason Algorithm 7 was chosen to initialise Maubach's method in preference to simply using two compatible triangles. Algorithm 7 enables fast initial triangle selection by creating virtual buckets. Therefore initial triangle selection can be achieved by finding a coarse triangle in T_0 containing $\mathbf{y}$; this is performed by `InitialTriangle` in Algorithm 14.

Algorithm 14: `InitialTriangle`

Input: $N_e \in \mathbb{N}$ and $\mathbf{y} \in [0,1]^2$
Output: $i \in \{1, 2, \ldots, N_t\}$

1 $\delta \leftarrow 1/N_e$
2 $(x, y) \leftarrow \mathbf{y}$
3 $x' \leftarrow \max\{1, \lceil N_e\, x \rceil\}$
4 $y' \leftarrow \max\{1, \lceil N_e\, y \rceil\}$
5 $i \leftarrow 2x' - 1 + 2N_e(y' - 1)$
6 **if** $y > \delta(x' + y' - 1) - x$ **then**
7 | **return** $i + 1$
8 **else**
9 | **return** i

`InitialTriangle` finds a coarse triangle in T_0 that contains $\mathbf{y}$ in constant running time by exploiting the simple geometry of uniform grids and the numbering of triangles shown in Figure 2. First a square containing $\mathbf{y}$ is ascertained by counting the triangles that would be traversed if the search were to begin in $t_1 \in T_0$, followed by $t_2 \in T_0$, and so on: on line 5 of Algorithm 14, $2N_e(y' - 1)$ is the number of coarse triangles in each traversed row, and $2x' - 1$ is the minimum number of coarse triangles traversed from the left of the grid to a square containing $\mathbf{y}$. Then a coarse triangle containing $\mathbf{y}$ will be above or below the line connecting the upper left and lower right corners of a square containing $\mathbf{y}$; this test is performed on line 6 of Algorithm 14.

7.2 Triangle Containing the Query Point

After employing `InitialTriangle` to initialise point location, T is traversed until a $t \in T$ is found that contains the query point $\mathbf{y}$. This section is devoted to the examination of a walking method for traversing T that uses the orientation test shown in Figure 6 to determine the next step in the walk. The `PointLocation` routine (Algorithm 15) utilizes this walking method to find a t containing $\mathbf{y}$.

The objective of a walking method is to move closer to $\mathbf{y}$ with each step in the walk. Let t be the current triangle being tested for containing $\mathbf{y}$. `PointLocation` effectively bisects Ω along each edge of t and then steps into the first neighbour of t that belongs to the subdomain containing $\mathbf{y}$. The orientation test determines the subdomain containing $\mathbf{y}$: if t does not contain $\mathbf{y}$, then $\mathbf{y}$ will be to the right of one of the edges of t; see Figure 6.

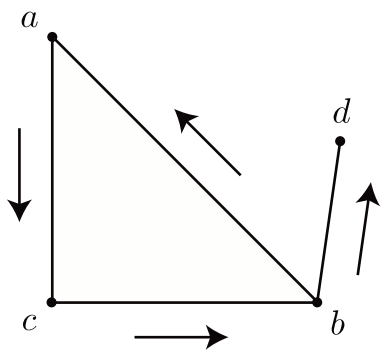


Figure 6: *The orientation test: does d lie on, to the left of, or to the right of the line defined by the triangle edge (a, b)? The arrows indicate the directions of the vectors used to perform the test. The test is applied to the other edges in the same manner.*

If $\mathbf{y}$ is not to the right of any edge of t, then t must contain $\mathbf{y}$ and `PointLocation` returns the index of t and terminates.

The orientation test is performed by establishing the sign of the determinant of a matrix with entries constructed from the edge and point being tested [Soukal, Malková & Kolingerová 2012]. For the example shown in Figure 6, this is equivalent to determining the sign of the third component of the cross product $(d - b) \times (a - b)$.[10] The orientation test is performed on line 14 of Algorithm 15.

The orientation test is known to produce incorrect results when $\mathbf{y}$ and the edge being tested are approximately colinear [Shewchuk 1997*a*]. A detailed examination of this issue is beyond the scope of this report; however the impact of an erroneous orientation test is examined here. Let the edge $\mathbf{e}$ of t be approximately colinear with $\mathbf{y}$, and assume the orientation test has just produced an incorrect result. Consider these three possibilities:

1. a wrong neighbour of t is stepped into, or
2. subsequent steps cycle between neighbouring triangles that do not contain $\mathbf{y}$, or
3. subsequent steps oscillate between a triangle and its neighbour that does contain $\mathbf{y}$.

The first problem is inconsequential as the outcome will simply be a somewhat longer walk. The second problem is solved by testing the edges of t in a random order for each step of the walk [Devillers, Pion & Teillaud 2002]. This is successful because cycles can only occur if `PointLocation` continues to erroneously step into the neighbour of t that shares $\mathbf{e}$: since $\mathbf{y}$ cannot be colinear with two edges of t simultaneously, $\mathbf{y}$ must be to the right of one edge that is not $\mathbf{e}$, and therefore testing this edge before $\mathbf{e}$ will halt the cycle. The third problem is solved by preventing `PointLocation` from returning to the triangle that was previously tested, which will also improve the efficiency of `PointLocation` [Devillers, Pion & Teillaud 2002]. However, this may result in `PointLocation` returning a triangle t' that does not contain $\mathbf{y}$. Even so, in this case $\mathbf{y}$ must be approximately on the edge of t'

[10] The points a, b and d are considered here to be 3-dimensional vectors in the xy plane.

shared with its neighbour that does contain $\mathbf{y}$, and the resulting numerical error may be tolerable for the application employing point location.

`PointLocation` requires the vectors used to perform the orientation test to have an anticlockwise orientation about the triangle being tested, as per Figure 6. The data structure $\mathcal{E}[i]$ is used to ensure the edges of t_i form vectors with the desired orientation, in an efficient manner.[11] The edges of all $t_i \in T$ are represented implicitly by $\mathcal{E}[i]$, where

$$\mathcal{E}[i] = \begin{cases} ((1,2),(3,1),(2,3)), & k(i) = 1 \text{ and } O(i) = 0 \\ ((2,1),(3,2),(1,3)), & k(i) = 1 \text{ and } O(i) = 1 \\ ((3,1),(2,3),(1,2)), & k(i) = 2 \text{ and } O(i) = 0 \\ ((1,3),(2,1),(3,2)), & k(i) = 2 \text{ and } O(i) = 1, \end{cases}$$

for $i \in \{1, 2, \ldots, N_t\}$. $\mathcal{E}[i]$ has the following attributes:

- $\mathcal{E}[i, j]$ represents the jth edge of t_i
- $\mathcal{E}[i, 1]$ represents the bisection edge of t_i with the remaining edges arranged anticlockwise about t_i
- $\mathcal{E}[i, j]$ for $j = 1, 2, 3$ (in order) are the positions of $\mathcal{T}[i, 1:3]$ containing references to nodes that enable vectors to be formed that rotate anticlockwise about t_i.

For example, consider the case $k(i) = 1$ and $O(i) = 0$. Then $\mathcal{N}[\mathcal{T}[i,1]] - \mathcal{N}[\mathcal{T}[i,2]]$, $\mathcal{N}[\mathcal{T}[i,3]] - \mathcal{N}[\mathcal{T}[i,1]]$ and $\mathcal{N}[\mathcal{T}[i,2]] - \mathcal{N}[\mathcal{T}[i,3]]$ (in order) are vectors that rotate anticlockwise about t_i.

An implementation of `PointLocation` is presented in Algorithm 15. `PointLocation` calls `InitialTriangle` and returns the index of a triangle containing the query point $\mathbf{y}$. Algorithm 15 has constant running time and requires insignificant additional memory to store $\mathcal{E}$. Recall that if $i =$ `InitialTriangle`$(N_\mathrm{e}, \mathbf{y})$, then the vertices of $t_i \in T$ are at most a distance of $\overline{h}$ from $\mathbf{y}$. Equivalently, a triangle containing $\mathbf{y}$ will be at most $\mathcal{O}(2^{M_L})$ steps away from t_i, where $M_L = \max_{j \in \{1,2,\ldots,N_t\}} L(t_j)$. Hence the number of steps taken by `PointLocation` is limited to 2^{M_L}. The orientation test is performed on line 14 of Algorithm 15.

Note that Algorithm 15 is only valid for $\Omega = [0, 1]^2$. Extending Algorithm 15 to allow for non-square domains and domains with holes is considered in Appendix A.

[11] $\mathcal{E}$ can also be used with the boundary marker β to efficiently determine the references to nodes on the boundary of Ω. This is particularly useful for a general Ω, which is considered in Appendix A.

Algorithm 15: `PointLocation`

Input: $N_\mathrm{e} \in \mathbb{N}$ and $\mathbf{y} \in \Omega$
Output: $i \in \{1, 2, \ldots, N_t\}$

$i_\mathrm{prev} \leftarrow 0$
$n \leftarrow 0$
$n_\mathrm{max} \leftarrow 2^{M_L}$
$B \leftarrow$ `True`
$i \leftarrow$ `InitialTriangle`$(N_\mathrm{e}, \mathbf{y})$
$t \leftarrow (\mathcal{N}[\mathcal{T}[i,1]], \mathcal{N}[\mathcal{T}[i,2]], \mathcal{N}[\mathcal{T}[i,3]])$
while B and $n \leqslant n_\mathrm{max}$ **do**
    $B \leftarrow$ `False`
    $E \leftarrow \mathcal{E}[i]$
    $P \leftarrow$ random permutation of $\{1, 2, 3\}$
    foreach $j \in P$ **do**
        $(v_1, v_2) \leftarrow \mathbf{y} - t[E[j,2]]$
        $(e_1, e_2) \leftarrow t[E[j,1]] - t[E[j,2]]$
        `// do the orientation test, but don't go back or step out of` Ω
        if $\mathcal{T}[i, j+3] \neq i_\mathrm{prev}$ and $\mathcal{T}[i, j+3] > 0$ and $\mathrm{sign}(v_1 e_2 - v_2 e_1) > 0$ **then**
            $B \leftarrow$ `True`
            $i_\mathrm{prev} \leftarrow i$
            $i \leftarrow \mathcal{T}[i, j+3]$
            $t \leftarrow (\mathcal{N}[\mathcal{T}[i,1]], \mathcal{N}[\mathcal{T}[i,2]], \mathcal{N}[\mathcal{T}[i,3]])$
            break
    $n \leftarrow n + 1$
return i

8 Conclusion

This report provides a comprehensive implementation of the unstructured mesh generation method of Maubach [1995], focussing on the case of a two-dimensional mesh on a square domain. An extension to this implementation that enables mesh generation on two-dimensional non-square domains and domains with holes is presented in Appendix A.

The implementation has the following desirable features. Mesh data structures were chosen to enable local bisection refinement to occur in constant running time and with minimal computations. A local refinement condition was derived that guarantees the local triangulation diameter of the refined mesh will obey a specified bound; hence the numerical error of computations on the mesh can be controlled while restricting the number of triangles. The mesh refinement algorithm was tested and shown to achieve the anticipated linear running time with respect to the number of triangles in the refined mesh. Employing a uniform triangulated grid to initialise Maubach's method creates virtual buckets. It follows that, for a mesh generated using Maubach's method, point location can be accomplished in constant running time without requiring additional memory.

Acknowledgements

During the preparation of this report I received invaluable feedback from Rodney Brown, Kevin McDonald, Michael Papasimeon and Josef Zuk.

References

Arnold, D. N., Mukherjee, A. & Pouly, L. (2000) Locally adapted tetrahedral meshes using bisection, *SIAM J. Sci. Comput.* **22**(2), 431–448.

Asano, T., Edahiro, M., Imai, H. & Iri, M. (1985) Practical use of bucketing techniques in computational geometry, *in Computational Geometry*, Vol. 2 of *Machine Intelligence and Pattern Recognition*, Elsevier, North Holland, pp. 153–195.

Cheng, S.-W., Dey, T. K. & Shewchuk, J. R. (2012) *Delaunay Mesh Generation*, Chapman and Hall/CRC Computer and Information Science Series, Chapman and Hall/CRC.

Devillers, O., Pion, S. & Teillaud, M. (2002) Walking in a triangulation, *International Journal of Foundations of Computer Science* **13**(2), 181–199.

Hjelle, Ø. & Dæhlen, M. (2006) *Triangulations and Applications*, Mathematics and Visualization, Springer-Verlag, Berlin.

Looker, J. R. (2013) *Globally Optimal Path Planning with Anisotropic Running Costs*, DSTO–TR–2815, Melbourne, Vic., Defence Science and Technology Organisation (Australia).

Maubach, J. M. (1995) Local bisection refinement for n-simplicial grids generated by reflection, *SIAM J. Sci. Comput.* **16**(1), 210–227.

Mücke, E. P., Saias, I. & Zhu, B. (1999) Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations, *Computational Geometry* **12**, 63–83.

Nelson, C. (2014) "Making Sense of CFD Grid Types", innovative-cfd.com, `http://www.innovative-cfd.com/cfd-grid.html`, accessed January 2015.

Shewchuk, J. R. (1997*a*) Adaptive precision floating-point arithmetic and fast robust geometric predicates, *Discrete and Computational Geometry* **18**, 305—363.

Shewchuk, J. R. (1997*b*) *Delaunay Refinement Mesh Generation*, PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

Shewchuk, J. R. (2012) *Combinatorial Scientific Computing*, Chapman and Hall/CRC Computational Science, Chapman and Hall/CRC, chapter 10.

Soukal, R., Malková, M. & Kolingerová, I. (2012) A new visibility walk algorithm for point location in planar triangulation, *in Advances in Visual Computing*, Vol. 7432 of *Lecture Notes in Computer Science*, Springer, Berlin Heidelberg, pp. 736–745.

Appendix A Domains other than $[0,1]^2$

The implementation of Maubach's method documented in the body of this report is only valid for $\Omega = [0,1]^2$. This implementation is now extended to allow for non-square domains and domains with holes, by covering the general domain Ω with a square domain, initialising Maubach's method using this square domain, and then removing the triangles not in Ω. Maubach's method can then be used to refine the triangles in Ω.

First consider $\Omega = [a,b]^2$. Let $D_\infty = b - a$ and $\mathbf{y}_c = (y_1, y_2)$ such that

$$\Omega = [y_1 - D_\infty/2, y_1 + D_\infty/2] \times [y_2 - D_\infty/2, y_2 + D_\infty/2]\,. \tag{A1}$$

Let $u : [0,1]^2 \to [a,b]^2$ where

$$u(\mathbf{y}) = D_\infty\,(\mathbf{y} - (0.5, 0.5)) + \mathbf{y}_c. \tag{A2}$$

The inverse of u is $u^{-1} : [a,b]^2 \to [0,1]^2$ where

$$u^{-1}(\mathbf{y}) = \frac{1}{D_\infty}\,(\mathbf{y} - \mathbf{y}_c) + (0.5, 0.5). \tag{A3}$$

In this case N_e and δ are given by:

$$N_\mathrm{e} = \left\lceil \frac{D_\infty}{\bar{\delta}} \right\rceil, \tag{A4}$$

$$\delta = \frac{D_\infty}{N_\mathrm{e}}. \tag{A5}$$

The following changes must be made to accommodate the case $\Omega = [a,b]^2$:

1. use the definition of N_e given by Equation (A4) in all algorithms
2. use the definition of δ given by Equation (A5) in all algorithms
3. immediately after line 4 of Algorithm 7, replace $\mathcal{N}$ with the result of applying the function u to each element of $\mathcal{N}$
4. replace $\mathbf{y}$ with $u^{-1}(\mathbf{y})$ on line 5 of Algorithm 15.

Now consider the general case where Ω is non-square, possibly with holes. Assume there exists a routine `DomainQ`$(\mathbf{y})$ that returns `True` if $\mathbf{y} \in \Omega$, or `False` otherwise, and determine a square domain as per Equations (A1) to (A5) that covers Ω, that is, if $\mathbf{y} \in \Omega$ then $\mathbf{y} \in [a,b]^2$. To accommodate the general case, make the above changes for a square domain and modify `InitialiseMesh` (Algorithm 7) as follows:

5. immediately after line 6 of Algorithm 7, call `FilterNodes` (Algorithm 16)
6. then call `FilterVertices` (Algorithm 17)
7. then call `FilterNeighbours` (Algorithm 18).

Finally,

8. replace line 5 of Algorithm 15 with $i \leftarrow \mathcal{M}_t\big[\texttt{InitialTriangle}\big(N_t, u^{-1}(\mathbf{y})\big)\big]$,

where $\mathcal{M}_t$ is constructed by Algorithm 17.

Algorithm 16: `FilterNodes`

Input: $\mathcal{N}$ generated by Algorithm 4
Result: nodes not in Ω are removed from $\mathcal{N}$, $\mathcal{M}_{\mathbf{x}}$ is constructed and $N_{\mathbf{x}}$ is updated

$j \leftarrow 1$
$\mathcal{M}_{\mathbf{x}} \leftarrow (1, 2, \ldots, N_{\mathbf{x}})$
$\widetilde{\mathcal{N}} \leftarrow (\mathcal{P}_{\mathcal{N}}[1], \ldots, \mathcal{P}_{\mathcal{N}}[N_{\mathbf{x}}])$
$n \leftarrow N_{\mathbf{x}}$
$N_{\mathbf{x}} \leftarrow 0$
foreach $i \in (1, 2, \ldots, n)$ **do**
    if `DomainQ`$(\mathcal{N}[i])$ **then**
        $N_{\mathbf{x}} \leftarrow N_{\mathbf{x}} + 1$
        $\widetilde{\mathcal{N}}[N_{\mathbf{x}}] \leftarrow \mathcal{N}[i]$
        $\mathcal{M}_{\mathbf{x}}[i] = j$
        $j \leftarrow j + 1$
    else
        $\mathcal{M}_{\mathbf{x}}[i] = 0$
$\mathcal{N} \leftarrow \widetilde{\mathcal{N}}[1 : N_{\mathbf{x}}]$

Algorithm 17: `FilterVertices`

Input: $\mathcal{T}_0^{\mathrm{V}}$ generated by Algorithm 5
Result: references to nodes not in Ω are removed from $\mathcal{T}_0^{\mathrm{V}}$, $\mathcal{M}_t$ is constructed and N_t is updated

$j \leftarrow 1$
$\widetilde{\mathcal{T}_0^{\mathrm{V}}} \leftarrow \big(\mathcal{P}_{\mathcal{T}}^0[1], \ldots, \mathcal{P}_{\mathcal{T}}^0[N_t]\big)$
$\mathcal{M}_t \leftarrow (1, 2, \ldots, N_t)$
$n \leftarrow N_t$
$N_t \leftarrow 0$
`// triangles that do not have all three vertices in` Ω `are removed`
foreach $i \in (1, 2, \ldots, n)$ **do**
    $V \leftarrow \big(\mathcal{M}_{\mathbf{x}}\big[\mathcal{T}_0^{\mathrm{V}}[i, 1]\big], \mathcal{M}_{\mathbf{x}}\big[\mathcal{T}_0^{\mathrm{V}}[i, 2]\big], \mathcal{M}_{\mathbf{x}}\big[\mathcal{T}_0^{\mathrm{V}}[i, 3]\big]\big)$
    if $V[1] > 0$ and $V[2] > 0$ and $V[3] > 0$ **then**
        $N_t \leftarrow N_t + 1$
        $\widetilde{\mathcal{T}_0^{\mathrm{V}}}[N_t] \leftarrow V$
        $\mathcal{M}_t[i] = j$
        $j \leftarrow j + 1$
    else
        $\mathcal{M}_t[i] = \beta$
$\mathcal{T}_0^{\mathrm{V}} \leftarrow \widetilde{\mathcal{T}_0^{\mathrm{V}}}[1 : N_t]$

Algorithm 18: `FilterNeighbours`

Input: $\mathcal{T}_0^{\mathrm{N}}$ generated by Algorithm 6
Result: references to triangles not in Ω are removed from $\mathcal{T}_0^{\mathrm{N}}$

$k \leftarrow 0$
$n \leftarrow \left|\mathcal{T}_0^{\mathrm{N}}\right|$
$N \leftarrow (0, 0, 0)$
$\widetilde{\mathcal{T}_0^{\mathrm{V}}} \leftarrow \left(\mathcal{P}_{\mathcal{T}}^0[1], \ldots, \mathcal{P}_{\mathcal{T}}^0[n]\right)$
// $n \neq N_t$ when `FilterNeighbours` is called
foreach $i \in (1, 2, \ldots, n)$ **do**
    if $\mathcal{M}_t[i] > 0$ **then**
        foreach $j \in (1, 2, 3)$ **do**
            if $\mathcal{T}_0^{\mathrm{N}}[i, j] > 0$ **then**
                $N[j] \leftarrow \mathcal{M}_t\left[\mathcal{T}_0^{\mathrm{N}}[i, j]\right]$
            else
                $N[j] \leftarrow \mathcal{T}_0^{\mathrm{N}}[i, j]$
        $k \leftarrow k + 1$
        $\widetilde{\mathcal{T}_0^{\mathrm{N}}}[k] = N$
// now $k = N_t$
$\mathcal{T}_0^{\mathrm{N}} \leftarrow \widetilde{\mathcal{T}_0^{\mathrm{N}}}[1 : k]$

Appendix B Other Cases for Bisecting Triangles and Updating their Neighbours

Algorithm 19: `BisectBoundaryTriangle`

Input: $i \in \{1, 2, \ldots, N_t\}$ such that $\mathcal{T}[i, 4] < 0$
Result: $\mathcal{T}[i]$ is updated; $\mathcal{N}[N_{\mathbf{x}} + 1]$ and $\mathcal{T}[N_t + 1]$ are created

```
CheckMemory( )
V ← T[i, 1 : 3]
N_x ← N_x + 1
k(i) ← 2 − (T[i, 8] mod 2)
N[N_x] ← 0.5 (N[V[1]] + N[V[k + 1]])
N_t ← N_t + 1
O ← (T[i, 7] + 1) mod 2
T[i, 8] ← T[i, 8] + 1
if k = 1 then
    T[i, 1] ← V[1]
    T[i, 2] ← N_x
    T[i, 3] ← V[3]
    T[N_t] ← (V[2], N_x, V[3], 0, 0, 0, O, T[i, 8])
else
    T[i, 1] ← V[1]
    T[i, 2] ← V[2]
    T[i, 3] ← N_x
    T[N_t] ← (V[2], V[3], N_x, 0, 0, 0, T[i, 7], T[i, 8])
switch (k(i), T[i, 7]) do
    case (1, 0) or (2, 1)
        UpdateBoundaryNeighbours1(i)
    otherwise
        UpdateBoundaryNeighbours2(i)
```

Algorithm 20: `ReplaceTriangleNeighbour`

Input: $i, j, k \in \{1, 2, \ldots, N_t\}$
Result: t_i, which was a neighbour of t_j, is updated to be a neighbour of t_k

```
if i > 0 then
    if T[i, 4] = j then
        T[i, 4] ← k
    else if T[i, 5] = j then
        T[i, 5] ← k
    else
        T[i, 6] ← k
```

Algorithm 21: `UpdateNeighbours2`

Input: $i \in \{1, 2, \ldots, N_t - 2\}$ such that $(k(i), \mathcal{T}[i, 7], \mathcal{T}[i_b, 7]) = (1, 1, 0)$ or $(2, 0, 1)$
Result: $\mathcal{T}[i, 4:6]$ and $\mathcal{T}[i_b, 4:6]$ are updated; $\mathcal{T}[N_t - 1, 4:6]$ and $\mathcal{T}[N_t, 4:6]$ are created

$\bar{i} \leftarrow N_t - 1$
$\bar{i}_b \leftarrow N_t$
$i_b \leftarrow \mathcal{T}[i, 4]$
$N \leftarrow \mathcal{T}[i, 5:6]$
$N_b \leftarrow \mathcal{T}[i_b, 5:6]$

$\mathcal{T}[i, 4] \leftarrow N[2]$
$\mathcal{T}[i, 5] \leftarrow i_b$
$\mathcal{T}[i, 6] \leftarrow \bar{i}$

$\mathcal{T}[i_b, 4] \leftarrow N_b[1]$
$\mathcal{T}[i_b, 5] \leftarrow \bar{i}_b$
$\mathcal{T}[i_b, 6] \leftarrow i$

$\mathcal{T}[\bar{i}, 4] \leftarrow N[1]$
$\mathcal{T}[\bar{i}, 5] \leftarrow i$
$\mathcal{T}[\bar{i}, 6] \leftarrow \bar{i}_b$

$\mathcal{T}[\bar{i}_b, 4] \leftarrow N_b[2]$
$\mathcal{T}[\bar{i}_b, 5] \leftarrow \bar{i}$
$\mathcal{T}[\bar{i}_b, 6] \leftarrow i_b$

`ReplaceTriangleNeighbour`$(N[1], i, \bar{i})$
`ReplaceTriangleNeighbour`$(N_b[2], i_b, \bar{i}_b)$

Algorithm 22: `UpdateNeighbours3`

Input: $i \in \{1, 2, \ldots, N_t - 2\}$ such that $(k(i), \mathcal{T}[i, 7], \mathcal{T}[i_b, 7]) = (1, 0, 1)$ or $(2, 1, 0)$
Result: $\mathcal{T}[i, 4:6]$ and $\mathcal{T}[i_b, 4:6]$ are updated; $\mathcal{T}[N_t - 1, 4:6]$ and $\mathcal{T}[N_t, 4:6]$ are created

$\bar{i} \leftarrow N_t - 1$
$\bar{i}_b \leftarrow N_t$
$i_b \leftarrow \mathcal{T}[i, 4]$
$N \leftarrow \mathcal{T}[i, 5:6]$
$N_b \leftarrow \mathcal{T}[i_b, 5:6]$

$\mathcal{T}[i, 4] \leftarrow N[1]$
$\mathcal{T}[i, 5] \leftarrow \bar{i}$
$\mathcal{T}[i, 6] \leftarrow i_b$

$\mathcal{T}[i_b, 4] \leftarrow N_b[2]$
$\mathcal{T}[i_b, 5] \leftarrow i$
$\mathcal{T}[i_b, 6] \leftarrow \bar{i}_b$

$\mathcal{T}[\bar{i}, 4] \leftarrow N[2]$
$\mathcal{T}[\bar{i}, 5] \leftarrow \bar{i}_b$
$\mathcal{T}[\bar{i}, 6] \leftarrow i$

$\mathcal{T}[\bar{i}_b, 4] \leftarrow N_b[1]$
$\mathcal{T}[\bar{i}_b, 5] \leftarrow i_b$
$\mathcal{T}[\bar{i}_b, 6] \leftarrow \bar{i}$

`ReplaceTriangleNeighbour`$(N[2], i, \bar{i})$
`ReplaceTriangleNeighbour`$(N_b[1], i_b, \bar{i}_b)$

Algorithm 23: `UpdateNeighbours4`

Input: $i \in \{1, 2, \ldots, N_t - 2\}$ such that $(k(i), \mathcal{T}[i, 7], \mathcal{T}[i_b, 7]) = (1, 1, 1)$ or $(2, 0, 0)$
Result: $\mathcal{T}[i, 4:6]$ and $\mathcal{T}[i_b, 4:6]$ are updated; $\mathcal{T}[N_t - 1, 4:6]$ and $\mathcal{T}[N_t, 4:6]$ are created

$\bar{i} \leftarrow N_t - 1$
$\bar{i}_b \leftarrow N_t$
$i_b \leftarrow \mathcal{T}[i, 4]$
$N \leftarrow \mathcal{T}[i, 5:6]$
$N_b \leftarrow \mathcal{T}[i_b, 5:6]$

$\mathcal{T}[i, 4] \leftarrow N[2]$
$\mathcal{T}[i, 5] \leftarrow \bar{i}_b$
$\mathcal{T}[i, 6] \leftarrow \bar{i}$

$\mathcal{T}[i_b, 4] \leftarrow N_b[2]$
$\mathcal{T}[i_b, 5] \leftarrow \bar{i}$
$\mathcal{T}[i_b, 6] \leftarrow \bar{i}_b$

$\mathcal{T}[\bar{i}, 4] \leftarrow N[1]$
$\mathcal{T}[\bar{i}, 5] \leftarrow i$
$\mathcal{T}[\bar{i}, 6] \leftarrow i_b$

$\mathcal{T}[\bar{i}_b, 4] \leftarrow N_b[1]$
$\mathcal{T}[\bar{i}_b, 5] \leftarrow i_b$
$\mathcal{T}[\bar{i}_b, 6] \leftarrow i$

`ReplaceTriangleNeighbour`$(N[1], i, \bar{i})$
`ReplaceTriangleNeighbour`$(N_b[1], i_b, \bar{i}_b)$

Algorithm 24: `UpdateBoundaryNeighbours1`

Input: $i \in \{1, 2, \ldots, N_t - 1\}$ such that $(k(i), \mathcal{T}[i, 7]) = (1, 0)$ or $(2, 1)$
Result: $\mathcal{T}[i, 4:6]$ is updated; $\mathcal{T}[N_t, 4:6]$ is created

1 $\bar{i} \leftarrow N_t$
2 $\beta \leftarrow \mathcal{T}[i, 4]$
3 $N \leftarrow \mathcal{T}[i, 5:6]$
4 $\mathcal{T}[i, 4] \leftarrow N[1]$
5 $\mathcal{T}[i, 5] \leftarrow \bar{i}$
6 $\mathcal{T}[i, 6] \leftarrow \beta$
7 $\mathcal{T}[\bar{i}, 4] \leftarrow N[2]$
8 $\mathcal{T}[\bar{i}, 5] \leftarrow \beta$
9 $\mathcal{T}[\bar{i}, 6] \leftarrow i$
10 `ReplaceTriangleNeighbour`$(N[2], i, \bar{i})$

Algorithm 25: `UpdateBoundaryNeighbours2`

Input: $i \in \{1, 2, \ldots, N_t - 1\}$ such that $(k(i), \mathcal{T}[i, 7]) = (1, 1)$ or $(2, 0)$
Result: $\mathcal{T}[i, 4:6]$ is updated; $\mathcal{T}[N_t, 4:6]$ is created

1 $\bar{i} \leftarrow N_t$
2 $\beta \leftarrow \mathcal{T}[i, 4]$
3 $N \leftarrow \mathcal{T}[i, 5:6]$
4 $\mathcal{T}[i, 4] \leftarrow N[2]$
5 $\mathcal{T}[i, 5] \leftarrow \beta$
6 $\mathcal{T}[i, 6] \leftarrow \bar{i}$
7 $\mathcal{T}[\bar{i}, 4] \leftarrow N[1]$
8 $\mathcal{T}[\bar{i}, 5] \leftarrow i$
9 $\mathcal{T}[\bar{i}, 6] \leftarrow \beta$
10 `ReplaceTriangleNeighbour`$(N[1], i, \bar{i})$

THIS PAGE IS INTENTIONALLY BLANK

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA				1. CAVEAT/PRIVACY MARKING
2. TITLE Mesh Generation via Local Bisection Refinement of Triangulated Grids			3. SECURITY CLASSIFICATION Document (U) Title (U) Abstract (U)	
4. AUTHOR Jason R. Looker			5. CORPORATE AUTHOR Defence Science and Technology Organisation 506 Lorimer St, Fishermans Bend, Victoria 3207, Australia	
6a. DSTO NUMBER DSTO–TR–3095	6b. AR NUMBER AR 016–273	6c. TYPE OF REPORT Technical Report	7. DOCUMENT DATE June 2015	
8. FILE NUMBER 2015/1008324/1	9. TASK NUMBER DS 07/245	10. TASK SPONSOR CJOAD	11. No. OF PAGES 37	12. No. OF REFS 13
13. URL OF ELECTRONIC VERSION `http://www.dsto.defence.gov.au/publications/scientific.php`			14. RELEASE AUTHORITY Chief, Joint and Operations Analysis Division	

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for Public Release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS

No Limitations

18. DSTO RESEARCH LIBRARY THESAURUS

algorithms, computer science, geometry, numerical algorithms

19. ABSTRACT

This report provides a comprehensive implementation of an unstructured mesh generation method that refines a triangulated grid by locally bisecting triangles on their longest edge, until they satisfy a given local condition. The method is relatively simple to implement, has the capacity to quickly generate a refined mesh with triangles that rapidly change size over a short distance, and does not create triangles with small or large angles.